December 2019

Scaling Up Automated Verification: A Case Study and a Formalization IDE for Building High Integrity Software

Daniel Thomas Welch
Clemson University, dtw.welch@gmail.com

SCALING UP AUTOMATED VERIFICATION: A CASE STUDY AND A FORMALIZATION IDE FOR BUILDING HIGH INTEGRITY SOFTWARE

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Daniel Thomas Welch
December 2019

Accepted by:
Dr. Murali Sitaraman, Committee Chair
Dr. Wayne Goddard
Dr. John D. McGregor
Dr. Nigamanth Sridhar

Abstract

Component-based software verification is a difficult challenge because developers must specify components formally and annotate implementations with suitable assertions that are amenable to automation. This research investigates the intrinsic complexity in this challenge using a component-based case study. Simultaneously, this work also seeks to minimize the extrinsic complexities of this challenge through the development and usage of a formalization integrated development environment (F-IDE) built for specifying, developing, and using verified reusable software components.

The first contribution is an F-IDE built to support formal specification and automated verification of object-based software for the integrated specification and programming language RESOLVE. The F-IDE is novel, as it integrates a verifying compiler with a user-friendly interface that provides a number of amenities including responsive editing for model-based mathematical contracts and code, assistance for design by contract, verification, responsive error handling, and generation of property-preserving Java code that can be run within the F-IDE.

The second contribution is a case study built using the F-IDE that involves an interplay of multiple artifacts encompassing mathematical units, component interfaces, and realizations. The object-based interfaces involved are specified in terms of new mathematical models and non-trivial theories designed to encapsulate data structures and algorithms. The components are designed to be amenable to modular verification and analysis.

Acknowledgments

This research was funded in part by National Science Foundation grants CCF-0811748, CCF-1161916, and DUE-1022941. I would like to acknowledge the members of the research committee for their assistance and support, including: Wayne Goddard, John McGregor, Nigamanth Sridhar, and Murali Sitaraman.

I would also like to acknowledge members of the RSRG (RESOLVE Software Research Group) at Clemson and OSU for their many helpful suggestions. Special thanks are due in particular to Bill Ogden and Joan Krone for providing the original motivation for many of the components and mathematical developments used in this research. Lastly, I'd like to thank Murali, my advisor, for his patience, guidance, kindness, and encouragement through all my years at Clemson.

On a personal note, I would also like to acknowledge my parents Tom and Judith Welch as well as my sister Anna and her partner John for their humor and limitless supply of encouragement. This dissertation would have never seen the light of day without it.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter provides an overview of the entire dissertation. We discuss the origins of program specification and verification, review some of the seminal work in the field, and identify several key barriers that, in practice, preclude routine, 'mainstream' program specification and verification. The chapter concludes with an overview that outlines the scope and context of this work, specific contributions we make towards addressing the identified challenges, and a broad organizational outline for the remaining chapters.

1.1 Motivation and Goal

The ability to formally specify and automatically verify functional correctness of software with respect to its formal specification is an ideal long pursued in the area of formal methods specifically, and the computing research community in general. Indeed, the very idea of formal specification goes as far back as 1949 to an essay written by A. Turing titled *On Checking a Large Routine* [106]. In this short note, Turing essentially floats the idea of formal program specification and verification wherein users can write "definite assertions" which "when checked individually" can establish "correctness of the whole program".

Investigations into the formal foundations of program verification, however, began in earnest in the mid to late 1960's with the work of King [57] and Hoare [51]. Nearly fifty years after these

initial contributions through Hoare's now seminal issuing of the 'verifying compiler' grand challenge [52] in the early 2000's, incremental advances in automated theorem proving, software engineering, and programming languages have collectively helped bring the ideal of formally verified software closer. Unlike testing, which can only reveal defects in software—not prove their absence, full formal verification has the potential to guarantee that code behaves according to its given specification under all inputs and valuations.

There are several barriers that remain however to achieving the verified software initiative put forth by Hoare. The first is the simple fact that the challenge demands languages that (1) permit users to write formal behavioral specifications for their code in the form of pre/post conditions and loop invariants and (2) automatically prove that component[1] implementations and client code satisfy such specifications.

There are a number of practical considerations in tackling this initial language barrier. Aside from the fact that language designers are expected to design new languages with verification in mind (something not entirely likely to occur), the goal also demands the presence of powerful automated theorem provers—which, more often than not, must be developed concurrently. This not only makes the challenge dependent on the ever-evolving state of the art in the entirely separate field of automated theorem proving (or, ATP for short) [85, 75, 76], but it also demands that language designers work overtime to connect their specification and verification language frameworks to a broad and varied range of separately designed ATP systems (though in the next chapter we do examine one such effort that accomplishes this).

Perhaps the biggest barrier to overcome, however, lies within the mindset of software development community itself, which still considers the problem of automatically verifying software to be simply too difficult: citing either the daunting complexity of automated provers, or the sometimes deep mathematical insight needed in general to prove theorems. Nevertheless, the view that programs are necessarily difficult to verify seems counter-intuitive when one considers that most programmers—including those without extensive mathematical backgrounds—are able to intuitively reason about their code, and convince themselves and others that their programs work as

[1] The (overloaded) term *component*, when used in the context of this work, is intended to mean a formal interface with one or more interchangeable realizations.

intended.

This same sort of intuition is now routinely used by researchers and practitioners of formal methods to alleviate lingering concerns over foundational issues such as undecidability.[2] The intuition is that nearly all programs which fail to verify automatically do not do not fail for foundational reasons such as incompleteness or undecidability, but rather, for mundane ones such as unsuitable specifications or incorrect, flawed code.

One central thesis guiding this work then is that well-engineered software components (e.g., those that adhere to established software engineering principles such as abstraction, modularization, and reusability) will not only lead to verification conditions (VCs) that mirror the simplicity of the programmer's intuitive understanding of their code [59], but will also enable verification to scale to larger, component-based systems. While there has indeed been substantial progress in developing suitable abstractions for, and verifying relatively isolated linear data structures such as stacks, queues, sets, and lists, the question of how existing techniques for specification and verification scale when faced with larger, more inherently complex layered data structures remains largely unexplored territory.

And though specification and verification of the components involved in a system of any size will no doubt be a time consuming, difficult, and expensive activity, there are two key characteristics we employ to offset these difficulties.

- first is the notion of reuse; that is, that the system overall is engineered from well-designed, *reusable* components that effectively amortize the high cost of their verification across subsequent usages;

- second is the critical notion of modularity, which allows us to specify and verify each individual component in isolation—thus implying the correctness of the entire system when composed.

Scalability naturally gives rise to the final barrier: a general lack of tool support for writing high integrity software of this kind [10]. Traditional programming languages have for many

[2] Prior to Hoare's 2003 challenge [52], such concerns arguably helped contribute to a chilling effect on program verification research. See [27] and [36] for some influential examples.

years enjoyed the support of powerful Integrated Development Environments (IDEs) that provide everything from a centralized workbench for projects of all sizes, to powerful code navigation and completion features that enhance user productivity and ease cognitive burden. To this end, a new class of IDEs broadly termed 'Formalization IDEs' (or, F-IDEs) aim to similarly assist users in crafting formal specifications and ease interaction with some underlying proof system. And though F-IDEs by themselves cannot realistically be expected to entirely eliminate the high costs inherent in formal specifications and verification—when employed in conjunction with the above two characteristics, F-IDEs have the potential to become an invaluable companion for developers building formally specified, component-based systems.

1.2 Contributions

To explore the question of scalability of component-based software verification and the role tools play in the process, this work offers two primary contributions. The first is an F-IDE built to support formal specification and push-button verification of imperative programs written in RESOLVE [100]—an integrated programming and specification/modeling language.

The second contribution is a case study that encompasses the formalization of a more complex, layered, component-based software system designed to demonstrate the scalability of our approach to reusable component specification and verification. We discuss each of these in greater detail below.

1.3 A Formalization Integrated Development Environment

To effectively support components with layered implementations and nontrivial specifications, while also providing a solid foundation for a responsive F-IDE, this work necessitated a ground up re-engineering of the existing RESOLVE research compiler.

The legacy RESOLVE system, built over the course of nearly two decades by successive generations of researchers and students, had become error prone and extremely difficult to extend, change, and maintain due to systemic, far-reaching design issues at the core of the language's im-

plementation. Though efforts have been made in recent years to address several of these systemic flaws, e.g., through automated mechanisms to ease traversal of the language's abstract syntax tree (AST) [34], this work takes a more radical re-engineering approach to enable long term scalability and more substantial language case studies (such as the one appearing in this work).

The result of this re-engineering effort is a pragmatic, lightweight, and extensible research compiler that is built on top of a reusable ANTLR4 [89] grammar that eschews fully fledged ASTs in favor of automatically constructed concrete syntax trees (or, parse trees) that provide default (builtin) traversal mechanisms e.g., tree listeners and visitors.

The following is a summary of some concrete benefits that the re-engineered compiler offers, and the role each one plays in enabling the construction of a robust F-IDE for RESOLVE.

Scalable Property-Preserving Java Code Generation. To facilitate generation of executable code, this work contributes a code generator that takes (verified) RESOLVE code as input, and faithfully translates it to Java code for execution on the Java Virtual Machine (JVM). The re-engineered code generator is capable of handling larger RESOLVE projects that span multiple workspaces and is designed to be run with a single click from within our F-IDE for fast testing and prototyping purposes.

Better Error Handling and Reporting. To facilitate descriptive error and warning annotations within the editor of the F-IDE, it was necessary outfit the new compiler with a resilient error manager. As opposed to the original compiler, which would simply stop on the first instance of an error and propagate it up to the compiler's main entry-point, the new system is resilient in the sense that it is capable of continuing in the presence of numerous errors and reporting them back as they are discovered via an 'error listening' interface. The F-IDE relies on this critical functionality to markup errors and warnings in files throughout a user's project.

A Prototype Mathematical Type Checker. To provide users with improved feedback when writing formal specifications, this work includes the development of a new (prototype) mathematical type checker that ensures formulas are well-formed and well-typed prior to reaching the verifier.

The type system is designed to be extensible and includes preliminary support for user defined subtyping via new "recognition" construct.

Support for Mathematical Specifications. To permit abstraction and modular reasoning (i.e., the ability to reason about components in isolation, independent of any one particular realization), a critical feature of RESOLVE is its usage of strictly mathematical specifications that adhere to established (notational) conventions in mathematics. For a variety of practical and implementation reasons, the RESOLVE compiler for many years was constrained to ASCII-based specifications that attempted to approximate the appearance of non-standard (unicode) operators within specifications.

Mathematical vs. Programmatic Specifications. RESOLVE remains relatively unique in its treatment of formal mathematical-style specifications. Consider for example the following quote from [66], which goes on to list a large number of (still-ongoing) programming and specification language efforts that favor a program-like syntax for specifications:

> "[E]xperience ... indicates that a mathematical syntax for assertions ... which is different from the programming language's syntax is a barrier to use by programmers. Programmers seem more comfortable with an assertion language that is based on the programming language's own expression syntax."

While programmatic specifications are perhaps more approachable, our experience using RESOLVE in the classroom has not revealed this to be a barrier. Rather, we find students often respond well to the mathematical syntax and readily use it to express formal assertions (such as invariants) and to distinguish such assertions from executable code [31]. Further benefits of mathematical specifications are discussed in Chap. 5.

This work adds support to the language for a wide range of mathematical (UTF-8) unicode characters, and the F-IDE we've built is designed to make insertion of such characters into the editor easy via LaTeX-style commands (or, optionally, via a symbol browser panel). The language's grammar has also been significantly updated to handle flexible syntax in specifications that permits, e.g., arbitrary outfix and infix notations. These changes make specifications distinctly mathematical in appearance—thus providing users with a clear visual distinction between the executable code and abstract specifications.

A Modular Verification Condition Generator. In RESOLVE's verification process, the role of the verification condition (VC) generator is to transform executable code into a set of mathematical formulas that describe both what is necessary and sufficient to prove in order to establish functional correctness of a particular fragment of code. This is the topic of [48], where an early prototype of RESOLVE's VC generator is presented, as well as an initial set of mechanizable proof rules that formally describe how to transform each language construct into a mathematical assertion.

Representing VCs as Gentzen-style sequents, this work incorporates recent efforts to simplify resultant VCs with a revised set of proof rules presented in [101]. The revised VC generator is designed to be modular and employs an architecture that makes it easy for developers to augment the behavior of existing proof rules, and add entirely new ones.

Similar to existing functionality in RESOLVE's web-based IDE [24], the F-IDE we present annotates each line in the editor with relevant VCs corresponding to the statement(s) appearing on that particular line. This work however goes further by incorporating an interactive view for exploring (step-by-step) the derivation of VCs. This feature grants a level of verifier transparency and traceability to both researchers as well as 'serious non-experts' [39]—such as students studying concepts in formal program verification.

1.4 A Component-Based Case Study

The case study we use to exercise the features of the F-IDE and the reengineered compiler involves the construction of a layered component for prioritizing arbitrary entries. We then use the developed components to conduct a smaller case study involving the construction and partial verification of an OS-Task scheduling application inspired by [30].

The case study involves a variety of artifacts including concepts that capture interface contracts, extensions to concepts, implementations (or realizations) of concepts and their extensions, as well as mathematical theories that are employed in specification and verification of correctness. The various artifacts are listed and grouped in Fig. 1.1.

A line connecting two artifacts within the queue and prioritizing family boxes indicates an

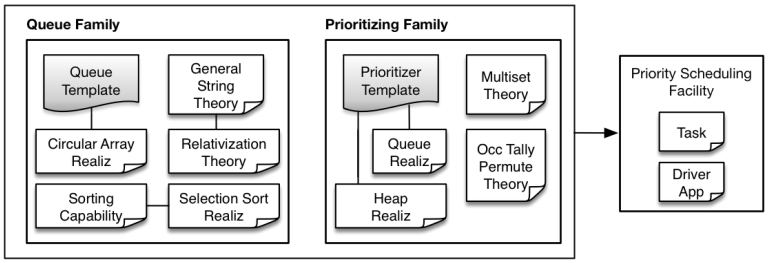


Fig. 1.1: A subset of different types of artifacts involved in the case study.

extends or realizes relationship. The first family of artifacts includes a bounded queue, a circular array realization, and a general sorting enhancement which we realize with a selection sorting realization. Interfaces and realizations in the queue family are specified in terms of a generalized version of RESOLVE's string theory. Several extensions to this theory are also presented, including one for polymorphic strings (i.e., strings relativized by the type of entries they contain), along with another describing string permutations.

The second family of artifacts introduces a prioritizing concept used for storing and retrieving generic entries based on a user provided ordering predicate. This part of the study introduces a new mathematical theory for multisets as well as a queue-based realization of the prioritizing concept. A second more experimental heap-based realization can be found in Appendix B.3.3. Several of the artifacts developed are used in the implementation of a small scheduling application (Fig. 1.1, far right).

The layered nature of the data structures and algorithms involved in the study represent a significant modeling and verification scalability challenge for component-based software. Some of the challenges include:

New Extensible Mathematical Theories. The study involves the addition of several small-to-moderately sized mathematical theories. Such theory developments—which are user-defined and extensible—provide the definitions, predicates, notations, and theorems that are used to simplify interface specifications and ultimately discharge (i.e. verify) any VCs arising from realization-level code. Some examples include theories corresponding to mathematical domains such as multisets, finite strings, ordering theory, and others. More well-established theories such as those for integers,

naturals, functions, and sets will also play a role.

Cross Cutting Verification. The realizations in the study, each of which employ their own mathematical models in conjunction with one or more user-defined theories, necessarily entails the ability to reason about more complex VCs. Automated systems that employ Satisfiability Modulo Theory (SMT) solvers—see Chap. 2, are tailored to operate only on a fixed set of highly specific theories, or decidable fragments thereof. Since this study involves multiple user defined theories, it will be instructive to study VCs arising from component interconnections—the proofs of which will inevitably require composition of results from multiple mathematical domains.

1.5 Outline

The rest of this work is organized as follows. Chap. 2 provides a broad overview of the spectrum of existing formal verification techniques, including a summary of several F-IDE supported efforts that most closely resemble the approach employed in this work. Chap. 3 provides background information on the existing RESOLVE language and walks through some examples. Chap. 4 provides a more theoretical treatment of RESOLVE's specification language and type system as well a tool for exploring VC derivations. Chap. 5 introduces the F-IDE, discusses its incorporation of the newest version of the compiler, and showcases its features. Chap. 6 presents the case study while Chap. 7 contains conclusions and future work.

Chapter 2

Overview of Program Verification Methods Tools and Techniques

This chapter provides an overview of three separate techniques to formal verification. We categorize each technique in the amount of interaction it requires from users, survey some of the notable efforts within each category, and summarize some achievements made possible through the use of such efforts. In the later half of the chapter, we discuss F-IDE backed verification efforts that most closely resemble the approach we employ in this work. The chapter concludes with a comparative summary of similar efforts and their supported F-IDE front-ends.

2.1 A Spectrum of Verification Techniques

Verification techniques can be broadly categorized along a spectrum with three distinct categories. Figure 2.1 provides an approximate illustration of these categories, and where they fall with respect to one another along this spectrum.

While each technique shown is centrally concerned with the problem of validating that a given system adheres to some formal specification or set of requirements, each one varies significantly in the amount of interaction required on the part of users. Correspondingly, as the names suggest, automatic methods typically require slightly less expertise from users, while interactive

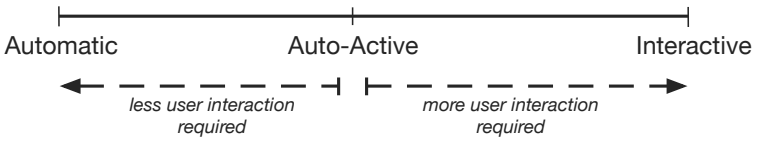


Fig. 2.1: Three categories of formal method techniques and their relationship to one another in terms of user-interaction

methods require more (but are usually considered more powerful and finely grained). This divergence between fully automatic (Sect. 2.2) and interactive (Sect. 2.3) is alternatively referred to in the literature as *lightweight* versus *heavyweight* methods, respectively. While such a binary taxonomy has worked well for many years, as we'll see in this chapter, new approaches actually tend to blur the line between the two—resulting in the need for a third category ('auto-active') which we discuss later in Sect. 2.5.

2.2 Automatic Model and Property Checking Techniques

On one end of the spectrum are techniques that attempt fully *automatic* verification with minimal to no user intervention. Such tools, which encompass techniques such as model checking and abstract interpretation, are used primarily as a means of 'checking' (i.e., proving) that a given concrete program or abstract model-based description adheres to some fixed, predefined set of properties [21].

There two classes of properties supported by most modern model checkers such as PRISM [63], UPAAL [64] and SPIN [53]. These include the following.

1. Safety properties which permit users to formally express invariants that must hold at the beginning and throughout the execution of a model—ensuring that certain erroneous states within the system are not reachable. For example, a verified safety property on the topic of (valid) memory can effectively 'rule out' the presence of *illegal* accesses such as null pointer dereferences, buffer overflows, or array indexing boundary errors.

2. Liveness properties which enable specification of realtime, reactive or parallel systems that take on a temporal dimension: for example, specifying that a given system will eventually

reach a certain (desirable) state, regardless of its starting configuration. These sorts of systems are common in real-world industrial applications and thus a variety of logics have been introduced to handle this style of specification including Computation Tree Logic (CTL) [20] and Linear Temporal Logic (LTL) [90].

While model checking tools in principle tend to support verification of both these properties, in practice different checkers tend to emphasize support for one or the other. For example, since UPAAL and PRISM are known for their handling of timed and probabilistic automata (respectively), their emphasis is naturally on liveness. On the other hand, since SPIN is built for model checking software (in particular, multithreaded C [114])—as opposed to hardware circuits—its focus tends more towards specification of safety properties [108].

Regardless of the properties being targeted, the typical workflow for using these types of tools starts with the construction of a higher level (abstract) model/architecture that captures one or more aspects of the system to be checked. Next, safety and/or liveness properties are specified and the model checker is run to determine whether or not the input model satisfies these properties.[1] If the properties are satisfied, the model is usually tossed out and reimplemented in a more concrete language, such as Java. Many tools include builtin mechanisms capable of automatically performing this conversion, so as to help rule-out errors introduced between the modeling and concrete implementation phases.

To guarantee full automation and termination when checking properties, input models have traditionally been restricted to be finite state[2]—which negatively impacts scalability. While a large amount of research has focused on improving this situation through better abstractions, modularity, and other ideas borrowed from the program verification realm [15], handling of infinite state models [22][3] and models that exhibit probabilistic behavior [60] nevertheless remain ongoing research directions. Additional work is also investigating avenues and techniques for automatically extract-

[1] This can be done in a variety of ways, for example, through simulation or symbolic model checking [78]

[2] This is mostly in an effort to avoid the *state explosion problem*—which says that the number of states in a model grows exponentially w.r.t. the size of its description

[3] In practice, infiniteness typically arises from two sources: reasoning about unbounded data structures (e.g., a queue without an upper bound on its length) and unbounded control structures (e.g., a program capable of spawning an unbounded number of threads) [2]

ing suitable (i.e., tractable finite) models from concrete source-code, though much work is needed for this to become generally applicable [2].

2.3 Interactive Techniques

On the far opposite end of the spectrum are strictly *interactive* efforts such as Coq [9], Isabelle [83], and (more recently) the Lean theorem prover [29]. These tools all require users to step into the proving process and manually construct proofs of correctness. Such tools typically pair expressive higher-order type theories and logics with powerful, general purpose theorem proving frameworks which—when combined—are well-suited to describing and verifying arbitrarily complex assertions and data structures. Several impressive efforts that underscore the power and mathematical flexibility of such approaches include machine checked proofs of mathematical results such as the Feit-Thompson odd order theorem [42], the central limit theorem [5], the Kepler 'cannonball-packing' conjecture [44, 43], and many others [41, 47, 33].

These same systems have been equally successful when used for program verification purposes. Some recent impressive, larger-scale efforts include the following.

- Chen et. al in [17] use Coq to verify a practical implementation of a posix file system that makes proven guarantees about file recoverability on abrupt system faults and failures (e.g. when the drive is unexpectedly unplugged, etc).

- Xavier Leroy in [73] presents a verified compiler named CompCert that is guaranteed (i.e., formally proven) to generate machine code that preserves the semantics of the source program. The generated code is efficient and compatible with existing embedded hardware platforms, thus making the system broadly applicable in practical, real-world contexts.

- Costanzo et. al in [26] propose a general methodology for verifying security of software written in C and assembly, then—building on the contribution of [73]—employ this to verify (in Coq) end-to-end security of a non-trivial operating system kernel that executes on verified CompCert x86 machine code.

The downside of such powerful, expressive interactive systems however lies in their inherent complexity and the steep learning curve they demand from users. Indeed, not only must users have knowledge of mathematical proofs and the ability to write, compose, and maintain detailed scripts for carrying them out, but must also—especially in the case of Coq and Lean—have a passing familiarity with the particulars of the type theory that underpins the chosen system, and its impact on proof processes [109, 25].

As a result, such tools currently tend to appeal only to a relatively small subset of computer scientists who are comfortable with intuitionistic (i.e. constructive) logic, knowledgeable of type-theory, and have a strong interest in functional-programming, specification, and/or formal methods in general.

Classical vs. Intuitionistic Logic. There are two primary, yet separate branches of formal logic: classical logic which admits the law of the excluded middle (i.e., that 'false' and 'not true' mean the same thing), and intuitionistic logic which rejects this law. For example, in an intuitionistic setting, it is not enough to say that $P \vee \neg P$ must be true—rather, there must be a proof of P (i.e. some construction of P) in order to justify this assertion.

This ability to "construct" proofs in a manner similar to constructing a program is one of the reasons intuitionistic logic is so attractive to computer scientists and is used as the basis for many interactive systems.

2.4 Auto-Active Techniques

Falling then squarely into the middle of the spectrum are *auto-active* verification efforts. Coined by Rustan Leino and Michal Moskał in 2010 [70], the term 'auto-active' is intended to describe efforts where users interact with the prover indirectly through formal annotations such as pre/post-conditions and loop invariants supplied through dedicated syntactic slots at the source code level. Some examples of languages drawn from this category include Dafny [68] and RESOLVE [100]: the language we ultimately employ in this work. Figure 2.2 illustrates the general style of interaction used in the vast majority of these approaches.

Specifically, auto-active approaches are guided first and foremost through the generation of

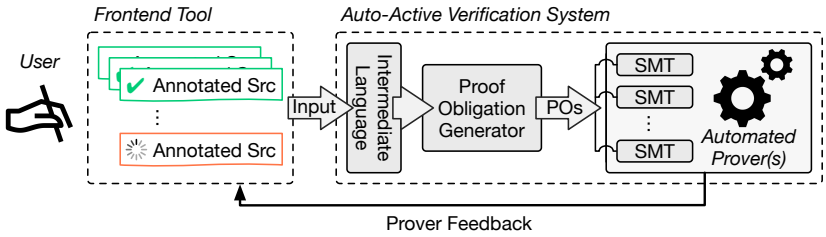


Fig. 2.2: A typical example of an auto-active tool's user feedback loop.

proof obligations from formally annotated user code. These obligations are then sent to a backend automated prover for verification.[4] If each proof obligation is successfully discharged, then the code is considered correct w.r.t. its formal specification. However, if one or more obligations fails to prove, users must indirectly interact with the prover by amending their code and/or specifications based on feedback provided by the verifier. This usually entails adding additional constraints, theorems, or lemmas that give the verifier the extra "hints" it needs to successfully (and autonomously) complete the required proofs. And since users do not play a direct role in guiding or carrying out concrete proof steps, these efforts are generally considered to be more approachable to average users. As a result, auto-active techniques naturally fall into the middle of the spectrum pictured in Figure 2.1, bridging the gap between fully automated and fully interactive approaches due to the tradeoff they make between automation, specification expressivity, and control over the proving process.

2.5 Overview of Closely Related Auto-Active Approaches

In this section we discuss some specific auto-active efforts within the spectrum of existing tools and techniques that are most closely related to our approach. Specifically, we focus on those that (i) tackle full functional verification of sequential programs and (ii) enjoy prominent frontend tool support in the form of one or more user-friendly graphical environments.

[4] In many cases the prover is actually a suite of *provers*, typically comprised of a number of Satisfiability Modulo Theory (SMT) solvers. The recent spike in popularity of this particular type of automated prover has actually been termed the *SMT revolution* by researchers in the field.

2.5.1 KeY

KeY [3] is a long running research project that comprises a collection of tools geared towards deductive verification of object-oriented programs written in Java. Specifically, KeY uses the Java Modeling Language (JML) [67] to express formal behavioral contracts through specially designated class and method level comments. For verification, JML annotated programs are first translated into a set of proof obligations (POs) that are expressed in a language called Java Card Dynamic Logic (or simply JavaDL—an extension of Hoare logic [51]), and are then sent to KeY's integrated backend prover. While this prover supports automation to a certain extent, in cases where it fails, the system is also capable of serving as an interactive proof assistant that allows users to systematically apply 'taclets' (i.e., *tactics*) to the current proof state—manually guiding the system towards the goal.

Though KeY can still be considered an auto-active effort, it is perhaps more accurate to label it an 'auto-interactive' approach as it supports both an auto-active workflow and a builtin interactive proof assistant that is reserved for more difficult proofs. Accordingly, KeY falls roughly halfway between the auto-active and interactive points of the spectrum shown in Fig. 2.1. While such a design conceivably permits more difficult assertions to be verified, it also demands more from users who—when faced with an unproven obligation—must decide whether to proceed along the traditional auto-active approach (e.g., by adding lemmas, additional specifications, etc), or simply attempt to dispatch the goal interactively—with all the additional expertise this requires.

2.5.1.1 User Interface Support

In terms of F-IDE support, KeY broadly supports two separate systems. The first is a standalone Java application that serves as the standard graphical user interface (GUI) for KeY—simply called KeY GUI. This environment is not a code editor, but rather, a general purpose 'viewer' that permits users to verify existing JML-annotated Java programs, explore resultant proof obligations, and generally control the manner in which the proof is carried out—such as where and when to apply a particular 'taclet' during a proof.

The second officially supported frontend is an extension that integrates KeY into Eclipse [49],

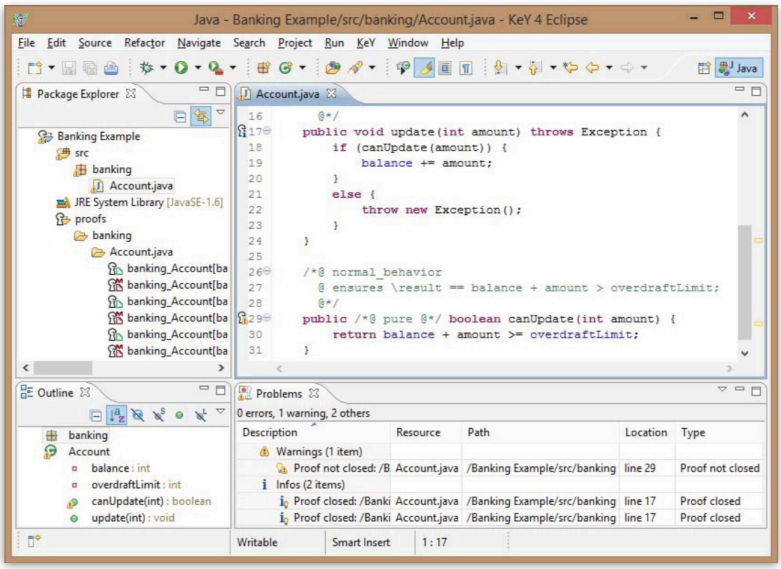


Fig. 2.3: KeY's GUI frontend with a proof obligation loaded.

and thus functions more as a traditional (code editing) development environment (see Fig. 2.3). Upon writing JML-annotated code—or changing existing code—the Eclipse plugin automatically invokes KeY's prover in the background (e.g., upon saving the document), and marks methods within the editor accordingly depending on whether or not the corresponding proof succeeded. The F-IDE tracks these annotations persistently across runs, visually indicating dependencies not yet fully verified. One notable benefit of KeY's integration into Eclipse is the fact that much of the existing (standard) Java IDE language functionality comes for free. Unfortunately, as of date, much of the functionality offered by the KeY GUI have not yet been integrated into the Eclipse environment—thus necessitating usage of two separate tools.

2.5.2 Dafny

Dafny [69] is an object oriented, imperative language in the spirit of C# from Microsoft Research that is designed from the ground up to support formal specification and automated verification. The language includes built-in syntactic slots for formal annotations such as pre/post conditions, loop invariants, and others—thus eliminating the need to retrofit them in through special comments (as in the case of KeY).

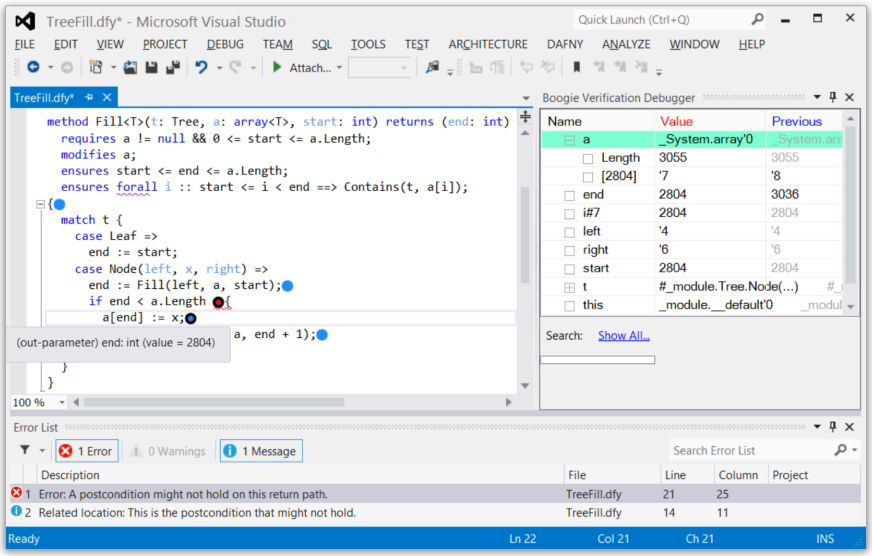


Fig. 2.4: The Dafny IDE in Visual Studio running the Boogie Verification Debugger.

Dafny makes no distinction between its specification and programming language. Rather, users simply mark functions and predicates accordingly as non-executable via a special 'ghost' keyword that signals to the verifier that these are strictly specificational in nature, and should not generate executable code. And in an effort to avoid features of mainstream languages that are known to complicate formal reasoning—such as uncontrolled referencing and aliasing [62]—Dafny's specification logic employs a notion of 'framing' (inspired by Dynamic Frames [56]) as a mechanism for reasoning about the heap and mutable effects [68].

Verification in Dafny works by first accepting a formally specified program and translating it into an intermediate verification language called Boogie [71]. Proof obligations are then generated from this intermediate representation and sent to Microsoft's popular automated theorem prover Z3 [28] for verification.

2.5.2.1 User Interface Support

F-IDE support for Dafny is provided through an extension to Microsoft Visual Studio [72], shown in Figure 2.5. Like KeY's official Eclipse plugin, the F-IDE for Dafny similarly provides users with design-time verifier feedback by continuously running the verifier in the background—

triggered by each new keystroke. To keep the system responsive, the IDE makes extensive use of caching (so proofs don't have to be recomputed unnecessarily) and multi-threading to allow multiple POs to be proven concurrently by different Z3 solver instances. The environment also includes the Boogie Verification Debugger (BVD) [65], which allows users to interactively explore states leading up to a failed assertion, such as a violated precondition. In an effort to mirror the way traditional program debuggers work (by inspecting concrete values of variables), such states are augmented with concrete values obtained through Z3 generated counterexamples.

2.5.3 Why3

Why3 [38] is a platform for deductive program verification that introduces a set of tools for specifying, implementing, and proving correctness of functional programs. For writing specifications, Why3 provides a language based on many-sorted first order logic with extensions for polymorphic types, pattern matching, inductive predicates, and higher order functions [12]. The (separable) implementation language, called WhyML,[5] natively supports slots for formal annotations and shares many of the same features as the specification language (such as pattern matching and polymorphic types) though not all (such as higher-order functions).

One characteristic that distinguishes Why3 from many other auto-active approaches is its support for user defined theory modules that house and export axioms, definitions, predicates, and lemmas that aid in simplifying the presentation of formal specifications. Indeed, theories describing integers, lists, functions, trees, and many others that come standard with the tool have been used to great effect in the specification and verification of many examples [103] and competition challenge problems [13].

In terms of verification process, Why3 is a strictly auto-active verification effort. That is, the platform (unlike KeY) provides no native avenue for expressing fine-grained control over how proofs are carried out. Rather, Why3 instead serves as a general API capable of producing and translating proof obligations (and the theories they employ) into a format compatible with both

[5] A dialect of the ML functional programming language that is augmented with some traditional imperative constructs such as for-loops

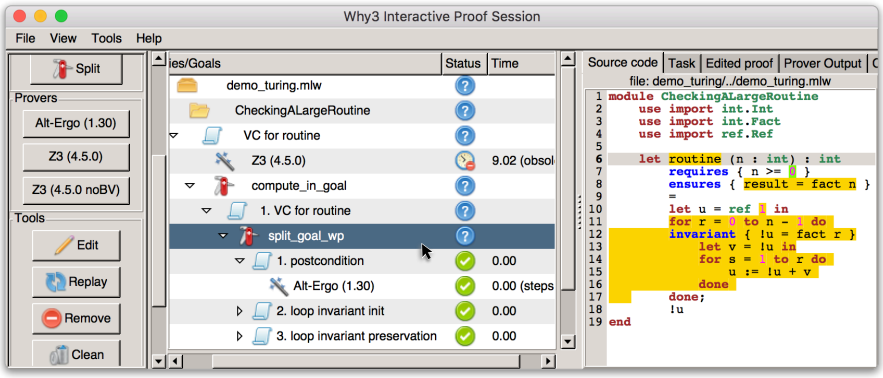


Fig. 2.5: The Why3 proof environment.

automatic provers (e.g., Z3 [28], Alt-Ergo [11], CVC3 [7], CVC4 [6], and SPASS [111]) as well as interactive ones (e.g., Coq [9], Isabelle [83], and PVS [86]). And though interactive provers are typically not known for their automated proving capabilities, those mentioned each provide a number of different 'auto' tactics which make them formidable auto-active proving backends [58].

More about Why3, including a formal proof of its soundness in Coq, details of its logical foundations, and an extensible mechanism for translating proof obligations into a format compatible across its many supported backend provers are discussed at length in [50, 37].

2.5.3.1 User Interface Support

Similar to KeY, Why3 comes standard with a GUI frontend for browsing project files and dispatching proof obligations. The environment also allows users to apply some proof-simplifying transformations such as, for example, splitting up a goal's internal conjuncts (thus producing separate 'subgoals') or expanding/replacing the application of a definition with its body.

For the purposes of regression testing, the environment also persistently tracks which goals have been verified thus far (along with their proofs) and allows users to 'replay' proofs of obsoletely verified goals to ensure that additions since the last successful verification attempt have not introduced new errors.

Similar to the GUI that comes with KeY, the Why3 GUI does not currently include the ability to edit code and/or specifications—only view files under consideration, and apply the afore-

mentioned goal transformations. Users must switch to a separate editor, make modifications, then switch back to the environment and manually reload each time a change is made.

2.5.4 AutoProof

AutoProof [104] is a static verifier that targets verification of functional correctness for formally specified, object-oriented Eiffel [79] programs. The system includes the usual annotations such as pre- and post conditions (i.e., contracts) as well as loop and class level (representation) invariants. Since the core language, Eiffel, is one of the first to pioneer and put into practice design-by-contract (DbC) principles [80], AutoProof naturally takes advantage of the language's native mechanisms and notation for expressing specifications. Such specifications, and the AutoProof programs they describe, are ultimately translated into Boogie which in turn employs Z3 for automated proving.

One defining characteristic of AutoProof's approach is its support for model-based specifications [92]. This style of specification permits users to abstractly model interfaces in an implementation-neutral way through the use of class level 'model' attributes[6] which separate specification from implementation. For purposes of abstraction, these attributes employ mathematical modeling library (MML) classes which correspond directly to mathematical concepts such as sets, bags, relations, sequences, functions, and others [99]. For extensibility, AutoProof provides so-called 'logic classes' as a mechanism for extending the specification language with new mathematical constructs which can then be wrapped in an MML class and used in specifications. To make these extensions compatible with Boogie, users must 'hook' their custom types to existing Boogie-supported types (such as sequences, Seq) via special `maps_to` clauses.

AutoProof's approach has been applied successfully to large program verification challenges and case studies—most notably the specification and verification of EiffelBase2, Eiffel's primary container library [95, 91]. The tool also been used successfully for contract based testing [93] and automated test generation [94] purposes.

[6] 'Attribute' here can be considered synonymous with (perhaps) more familiar terms such as 'field' or 'member'

2.5.4.1 User Interface Support

In terms of user-interfaces, AutoProof offers two choices. The first is a web-based environment named ComCom [115] that allows users to specify, implement, and verify (in a push-button manner) smaller, single class example programs. The environment itself—which provides only a minimal editor with syntax highlighting—lacks the ability to save work, and thus functions more as a 'sandbox' for quick experimentation without necessitating the overhead and hassle of a local installation.

The second environment, the Eiffel Verification Environment (EVE), is an open-source, desktop based IDE that integrates support for a variety of verification tools including AutoTest [81] (a tool for automatically generating tests from Eiffel contracts) and AutoProof. Built on top of EiffelStudio, the primary development platform for Eiffel, EVE permits users to easily verify their code in a push button manner and browse the results of a verification attempts within the IDE.

2.6 Discussion

A comparison of the auto-active approaches discussed throughout this chapter (including RESOLVE) is provided in Table 3.1. As shown, the majority of auto-active efforts target one or more third-party SMT solvers (in most cases, Z3) for the purposes of automated proving. While SMT solvers have earned a solid reputation for being quick and reliably automatic, they necessarily operate on first order formulas which constrains efforts targeting them to traditional first order logic, FOL (thus hindering expressivity). Though some efforts like Why3 and VeriFast attempt to remedy this by admitting certain higher order extensions such as polymorphic types and pattern matching, this ultimately requires the ability to translate higher-order POs down into an acceptable (and equivalent) first order form—which is a non-trivial process in general [14, 107]. Notable disadvantages include high potential for "impedance mismatches" when translating between IVLs [4], or (more commonly) when translating the constructs of the rich 'high level' specification language into the 'lower level' representation employed by a particular IVL [107]—or vice versa [39]. These mismatches in turn can complicate error reporting efforts, including VC feedback on failed verification

Effort	Specification Language (Ext.?)	Programming Language	Verification Technique	Supported Prover(s)	Targeted Logic
Dafny	Dafny (N)	C#-like	Auto-Active	Boogie/Z3	FOL
Why3	Why3 (Y)	WhyML	Auto-Active	Z3, CVC3-4, Coq, Yices1-2, ..	FOL
KeY	JML (N)	Java	Auto + Interactive	Z3 + KeY Proof Assistant	FOL
AutoProof	Eiffel (Y*)	Eiffel	Auto-Active	Boogie/Z3	FOL
VeriFast	VeriFast (N)	Java, C	Auto-Active	Z3	HOL
RESOLVE	RESOLVE (Y)	RESOLVE	Auto-Active	RESOLVE	HOL
Coq	Gallina (Y)	Coq	Interactive	-	HOL
Isabelle	Isar (Y)	Isabelle	Interactive + Auto Tactics	-	HOL
Lean	Lean (Y)	Lean	Interactive	-	HOL

Table 2.1: A summary of auto-active and interactive verification efforts. Here, FOL indicates that, foundationally, the tool is grounded in first order logic while HOL indicates a higher-order-logic

attempts.

First Order Logic vs. Higher Order Logic. Building on the basic declarative structure of propositional logic, first order logic adds the ability to express assertions involving a mixture of constants, functions, predicates, and quantification (universal and existential) over elements of some domain.

Higher order logic in turn builds on first order logic by admitting polymorphic types, lambda abstraction, quantification over variables of an arbitrary type (including functions), and the ability pass functions as arguments to other functions.

Among the efforts discussed, Why3 and to an extent, AutoProof, are the only other current autoactive efforts (excluding RESOLVE) that permit users to extend the underlying formal specification language with support for new domains. This functionality (signified through the 'Ext' in the second column) is handled natively in Why3 (and RESOLVE) through the use of theory modules, while AutoProof employs MML classes. Though both of these can serve as a suitable basis for writing new model based contracts, one notable downside of AutoProof's approach in particular is that users must link their proposed model class to a preexisting type encoded externally in Boogie. This not only tightly couples AutoProof to the capabilities of Boogie (with all of its hardcoded base

types), but also means that serious users who wish to add a new theory must write one externally in Boogie—requiring knowledge of two systems.

Regardless of support for extensible specifications, efforts based on existing languages generally require more complex specifications in order to cope with the complexities inherent in mainstream languages such unconstrained referencing and aliasing. For example, in order to effectively support verification of C and Java programs, VeriFast [54] employs a form of separation logic [96] to specify and reason about the shape of the heap. Though powerful and expressive, specifications based on separation logic are notably less abstract since they require specifiers to carefully consider low-level characteristics of the heap in their assertions. This extra detail in turn demands more expertise from users and generally results in a higher annotation overhead (both in terms specification line-count and time) than that required by model-based alternatives.

To help place the auto-active approaches discussed in context, we also include in Table 3.1 characteristics of several popular proof-assistants below the line in their own section. Of particular note is the fact that most employ expressive higher order logics, in addition to rich libraries of tactics that users may interactively apply to prove a given goal. This approach is reasonable, as full automation is not the primary objective of such systems.

Chapter 3

RESOLVE Background: The Language and Existing Toolchain

This chapter provides a high level overview of the RESOLVE language necessary to follow the rest of the dissertation. In particular, we walk through a comprehensive example of a component developed in RESOLVE which we discuss in the context of the language's existing web-based IDE. The intent of this chapter is to help the reader understand: (i) the language itself and its modeling capabilities, (ii) the web IDE's role as a research tool, and (iii) to provide the reader with a point of reference for distinguishing between the existing language and the new environment, tools, and language features that we introduce in this dissertation and make use of in later chapters.

3.1 Characteristics of a Language Designed for Verifying Software Components

Any effort that seeks to meet the challenge of verifying functional correctness of software demands a language and compiler with an extensible, flexible toolchain capable of scaling up to verification of component-based systems. The language we employ (and further develop) as part of this work is RESOLVE [100]: an integrated specification and programming language designed for building and verifying reusable software components.

There are a number of characteristics that make RESOLVE ideal for such a task, including:

- **Integrated Specificational Capabilities.** The language enforces a strict separation between abstract, mathematical model-based specifications used in interfaces and the executable code used in implementations. This separation is crucial as it keeps interface specifications free of implementation bias and permits modular reasoning—i.e. the ability to reason exclusively in terms of high level specifications without needing to consider implementation details.

- **Extensible Mathematics.** When considering the spectrum of software that one might seek to specify and verify, it is unlikely that only a handful of pre-defined mathematical models will suffice. As such, RESOLVE permits the addition of new structure-appropriate mathematical theories and theory-extensions. Such developments, which are suitable for describing arbitrary domains, permit users to write succinct specifications and enables automated reasoning. And though construction of such developments is expensive, this can be offset by reusing existing theories whenever possible.

- **Reusable Concepts and Components.** Concepts that combine generics with abstract contracts enables the development of highly reusable, decoupled components that support multiple realizations. This is critical as verified software is expensive to achieve, and thus should be reusable.

- **Clean Semantics.** The language restricts the harmful affects of aliasing, uncontrolled referencing, and mutation through a "clean semantics" [61]. This simplifies reasoning by ensuring: (1) that effects of code is kept local to a restricted subset of the program's overall state space and (2) that only variables explicitly named in such a subset are permitted to change.

3.2 The RESOLVE Verification System

The RESOLVE compiler is open source and is available online at: https://github.com/ClemsonRSRG. At the time of writing, there are two distinct versions of the compiler that share similar grammars and roughly the same compilation pipeline. The first is a more experimental version

developed as part of this work that supports the newer F-IDE and and the revised feature-set outlined in Sect. 1.3. The second (stable) release supports the web-based IDE and has been widely utilized in the software engineering curriculum at Clemson and elsewhere for teaching formal reasoning and component-based software design principles [23, 55, 31].

Figure 3.1 provides an overview of the general architecture and compilation pipeline shared by both systems. The yellow highlight indicates that we employ the web-based IDE (and hence the older version of the RESOLVE compiler) throughout the remainder of this chapter.

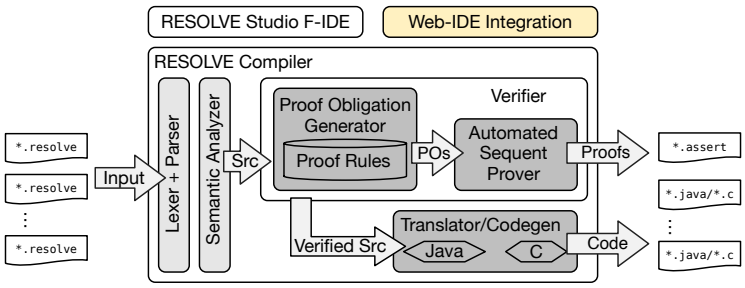


Fig. 3.1: High level architecture for the RESOLVE system and its associated compilation pipeline

The compilation process adheres to a relatively traditional setup consisting of lexing, parsing, semantic analysis, and code generation—with the notable addition of an automated verifier. The verifier, upon receiving suitably annotated source-code, first generates a collection of verification conditions (VCs) that are both necessary and sufficient for proving correctness of code w.r.t. some formal specification.

Each VC is represented as a *sequent* of the form:

$$\varphi_1, \cdots, \varphi_m \vdash \psi_1, \cdots, \psi_n$$

where m and n are non-negative integers and $\varphi_1, .., \varphi_m$, $\psi_1, .., \psi_n$ denote sets of well-formed-formulas (wffs) within a given sequent's *antecedent* and *succeedent*, respectively. The verification condition (VC) generator is responsible for generating such sequents and adding wffs to them from different specification contexts—as dictated by RESOLVE's sound and (relatively) complete set of proof rules. Semantically, each sequent adheres to the usual interpretation, $\bigwedge_{i=1}^{m} \varphi_i \vdash \bigvee_{j=1}^{n} \psi_j$ where the universal conjunction of all wffs on the left entails ($\vdash$) the universal disjunction of all wffs on the

right. As a shorthand, we frequently refer to a generic sequent as $\Gamma \vdash \Delta$ where implicitly Γ is a set of wffs representing the antecedent and Δ is a set of wffs representing the succeedent.

After VC generation, each sequent is sent off to RESOLVE's in-house congruence closure prover for verification [45, 82]. As illustrated in Fig. 3.2, a formally annotated program meets its specification when the verifier is able to automatically prove all VCs. If a VC fails to prove, this suggests a flaw in either the code or specification.

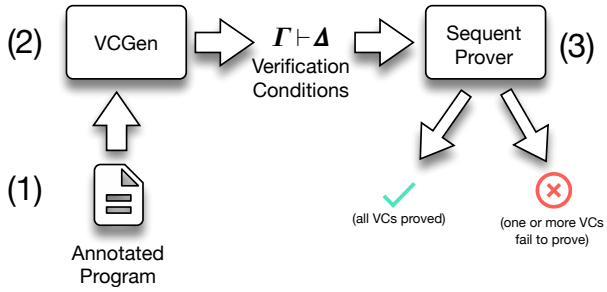


Fig. 3.2: A closer look at the steps in RESOLVE's verification pipeline; here the numbers identify the sequence of these steps

In Sect. 3.3 we discuss where VCs come from and the process by which users (both novices and experts) use them to identify which fixes are needed at the source code or specification level—which is why they must be human readable.

3.3 Example: Developing a Queue Concept and Components

In this section, using the example of a bounded queue concept, we give a concrete illustration of the language features described in Sect. 3.1 as well as a demonstration of the functionality offered by RESOLVE's existing web-based IDE.[1]

3.3.1 Relationship-Centered Artifact Management

Unlike traditional IDEs, which organize a user's workspace according to the underlying file-system, the web-based interface enforces an organization based on component relationships, shown in Fig. 3.3.

[1] http://resolve.cs.clemson.edu/teaching

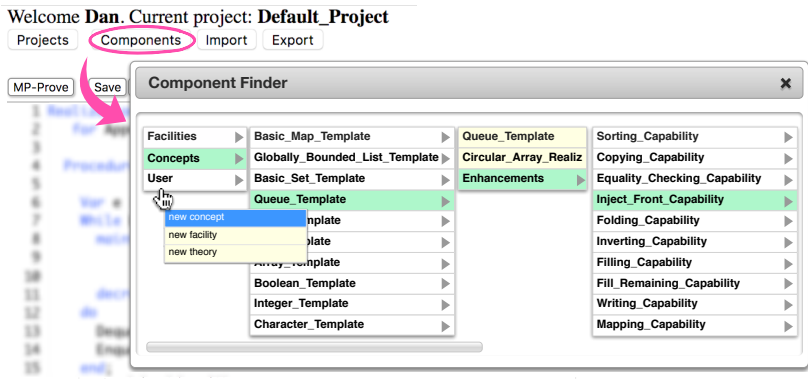


Fig. 3.3: The web-IDE's component browser

In particular, **Concepts** contains interface specifications for a variety of standard components ranging from lists and maps to sets. Upon selection of a specific concept, a new list reveals (in yellow) the concept interface itself and its realizations. This tab also contains **Enhancements** that offer additional functionality that can be layered on top of the base concept interface (enhancements are discussed further in Sect. 3.4). Finally, **Facilities** are general client programs that compose and use some combination of the components mentioned while **User** persistently stores any user-created artifacts.

New users can create an account and log into the environment. Doing so allows one to edit existing, "standard" components and create entirely new ones by using right clicks within the component-browser. Use of right clicks at different levels within the browser hierarchy opens a prompt that allows users to create modules that are appropriate for that particular level (e.g., on the outermost level, users can create only concepts, facilities, or theories).

3.3.2 Concept Specification

Every RESOLVE component has a formal interface specification. Even typically built-in objects such as arrays, integers, and pointers have their behavior specified by interfaces called concepts. An example concept interface, `Queue_Template`, is presented below in Fig. 3.4.

This particular specification is parameterized by a generic type `Entry` and an integer `Max_Length` that places an upper bound on the maximum number of entries a queue can hold. The **`evaluates`** parameter mode preceding `Max_Length` indicates that the actual bound passed to

RESOLVE / Web IDE

```
Concept Queue_Template (type Entry; evaluates Max_Length : Integer);
  uses Basic_String_Theory;
  requires 1 <= Max_Length;

  Type family Queue is modeled by Str(Entry);
    exemplar Q;
    constraints |Q| <= Max_Length
    initialization
       ensures Q = Empty_String

  Operation Enqueue (alters e : Entry; updates Q : Queue);
    requires |Q| + 1 <= Max_Length;
    ensures Q = #Q o <#e>;

  Operation Dequeue (replaces e : Entry; updates Q : Queue);
    requires |Q| /= 0;
    ensures #Q = <e> o Q;

  Operation Swap_First_Entry (updates e : Entry;
                              updates Q : Queue);
    requires |Q| /= 0;
    ensures e = DeString(Prt_Btwn(0, 1, #Q)) and
          Q = <#e> o Prt_Btwn(1, |#Q|, #Q);

   /* ... remaining operations omitted for brevity ... */
end Queue_Template;
```

RESOLVE / Web IDE

Fig. 3.4: An abstract specification for a bounded queue concept

the concept may be an integer-valued expression, and that it is to be evaluated.

The **uses** line that immediately follows gives the specification access to a mathematical theory of strings called `Basic_String_Theory`, which contains a number of useful definitions intended to help specifiers author succinct specifications for a given ADT. Theories naturally also contain theorems involving these definitions for use in verification, a point we return to in Sect. 3.3.3.

Next, the **Type family** declaration introduces a collection of abstract types called `Queues` that are modeled by strings of entries. The **exemplar** clause that immediately follows can be thought of as an example queue that acts as a representative to the ADT's family. Usage of the

exemplar can be seen immediately below where we use it to assert that not *all* strings can be models of valid queues, but rather, only those of length Max_Length or less.

The **initialization** clause **ensures** that all queues are guaranteed to be empty when initialized, which is to say—drawing comparisons to programming—it specifies an (abstract) constructor.

Now that the framework for our mathematical model is specified, we may now define some common operations on Queues such as Enqueue and Dequeue. Each of these operations is formalized by a pre- and post-condition that communicates (i) what must hold prior to a call (i.e. the **requires** clause), and (ii) a postcondition that specifies what must hold after (i.e. the **ensures** clause). For example, the **ensures** clause of Enqueue can be stated in English as follows:

> *"the outgoing value of Q is equal to the #-denoted incoming queue concatenated with the singleton string containing the incoming value of entry e."*

As with the parameters to the concept, each formal parameter to an operation is preceded by a specification mode that makes explicit what effect the operation will have on the parameter in question. Table 3.1 below summarizes RESOLVE's supported parameter modes.

Parameter mode	**Meaning**	**Valid realization alts.**
alters *x*	#*x* is meaningful, but its outgoing value is undefined	**clears**
replaces *x*	#*x* is meaningful, but is replaced with another meaningful value specified in the **ensures** clause	**clears**
updates *x*	#*x may* be meaningful, but will be changed to a meaningful value specified in the **ensures** clause	**clears**, **restores**, **preserves**
evaluates *x* : Ty	*x* can be an arbitrary expression, assuming its type is equivalent to (or a subtype of) Ty	–
preserves *x*	*x* cannot be changed at all, only read	–
restores *x*	*x* may change, however $x = \#x$ must hold at the end	**preserves**
clears *x* : Ty	*x* is reset to Ty's initial value	–

Table 3.1: A summary of supported specification parameter modes; here, #*x* refers to the "incoming value" of a parameter *x* (outgoing parameters implicitly drop the #). The rightmost column summarizes valid (overlapping) alternative modes that can be used in implementations

The mode specified can have subtle impacts on the specification and underlying performance of an operation's implementation. For example, a formal parameter with a highly permissive mode such as `alters` makes no guarantees about its outgoing value, and thus provides implementers with the most flexibility. Under this mode, implementations can avoid the need to copy either references (which causes unnecessary aliasing) or data representations (which can be expensive) [46]. Other modes such as `updates` by contrast are necessarily more all encompassing and thus can be used in lieu of other more specific modes with similar effects (such as `restores`, which can change the meaningful value passed as long as it is "updated" back to its original value by the end).

To assist new users, the web-based IDE provides explanations of various language keywords, including the parameter modes discussed. Fig. 3.5 illustrates how the web-based IDE communicates this through contextual tooltips that are brought up by double clicking relevant language keywords in the editor.

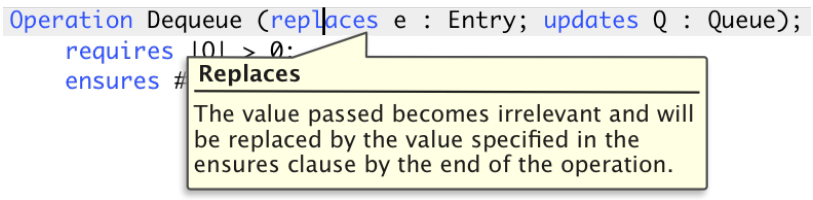


Fig. 3.5: Contextual tooltip explanations for specification keywords

In Chap. 5 we introduce and leverage a number of features afforded by our new F-IDE that provide users with additional assistance in both selecting parameter modes and eliminating common syntactic pitfalls when employing them.

3.3.3 Mathematical Support

The `Queue_Template` concept imports a mathematical précis module for (finite) strings called `Basic_String_Theory`. This module defines and exports a number of string-specific operators employed throughout the specifications—such as the string "constructor" `Str(..)`, which we use to formulate the model of the queue appearing in Fig. 3.4. Other common string opera-

tors include length (|..|), concatenation (o), Empty_String, and singleton (<..>). Additionally, Basic_String_Theory also exports a number of theorems and corollaries for use by RESOLVE's automated prover. A preview of these are shown in Fig. 3.6.

RESOLVE / Web IDE

```
/* Reversal Theorems */
Theorem Reverse_of_Singleton:
    Forall E : Entity, Reverse(<E>) = <E>;

Theorem Concatenation_Under_Reverse:
    Forall U, V: String, Reverse(U o V) = Reverse(V) o Reverse(U);

Theorem Reverse_Inverts_Itself:
    Forall S : String, Reverse(Reverse(S)) = S;

/* Permutation Theorems */
Theorem Identity_Permutation:
    Forall S : String, Is_Permutation(S, S);

Theorem Permutation_Lengths:
    Forall S, T : String, Is_Permutation(S, T) implies |S| = |T|;
```

RESOLVE / Web IDE

Fig. 3.6: A snippet of Basic_String_Theory

Taking a page from the world of object oriented programming's interface-implementation separation, readers will note that no proofs of the theorems listed appear at the **Precis** level. Rather, such theorems are proven offline and relegated to a separate module—as they provide a level of fine-grained information unneeded by most general users of the theory. This makes sense, as specifiers are generally seeking predicates and other operators that will assist them in shortening (or simplifying) their specifications—as opposed to intricate proofs of theorems and corollaries involving them.

RESOLVE's current library of theories is by no means complete, or, for that matter, exclusive to strings. Rather, any number of typical theories ranging from natural numbers and integers to more sophisticated ones such as trees and multisets have been developed and can ultimately be used in the specification of concepts—such as the queue showcased in this chapter [45].

RESOLVE / Web IDE

```
Realization Circular_Array_Realiz for Queue_Template;
VC  Type Queue is Record
VC      Contents : Array 0..Max_Length - 1 of Entry;
        Front, Length : Integer;
      end;
        conventions
VC           0 <= Q.Front <= Max_Length and
              0 <= Q.Length <= Max_Length;
    correspondence
VC      Conc.Q = Concatenation i : Integer,
            Q.Front <= i <= Q.Front + Q.Length - 1 implies
                  <Q.Contents(i mod Max_Length)>);

VC  Procedure Enqueue (alters e : Entry; updates Q : Queue);
VC    Q.Contents[(Q.Front + Q.Length) mod Max_Length] :=: e;
VC    Q.Length := Q.Length + 1;
    end Enqueue;

  /* ... Omitted for brevity ... */
end Circular_Array_Realiz;
```

RESOLVE / Web IDE

Fig. 3.7: A circular array realization for `Queue_Template`. The grey "VC" buttons indicate lines that generate one or more verification conditions

3.3.4 A Circular Array Realization

Once a concept has been specified, a realization must be provided before the data abstraction can be employed (and executed). While of course there could be any number of realizations of the same concept, in this chapter we focus exclusively on a "circular" array realization, shown in Fig. 3.7.

We represent a queue programmatically as a **`Record`** (analogous to a C struct) containing a `Contents` array as well as integers `Length` and `Front` which denote the current size and front of the queue, respectively.

A Note on Array Syntactic Sugar. One thing to note about the `Contents` array is that it's not a built-in type, but rather, is specified using the same concept machinery as any other component. Specifically the field declaration

```
Contents : Array 0 .. Max_Length - 1 of Entry;
```

is syntactic sugar for a factory (i.e. a **Facility**) that produces arrays of the specified type and length:

```
Facility Arr_Fac is
    Static_Array_Template(Entry, 0, Max_Length - 1)
      realized by ...
```

Thus, when the queue's `Contents` field is declared, the desugared form reads as follows: `Contents : Arr_Fac::Static_Array`, where '`::`' qualifies which module (or facility) the named symbol is being drawn from.

The array's operations are similarly sugared to employ a more familiar syntax, e.g.: accessing elements via `A[i]`, etc.

Restrictions on the programmatic representation of the queue are enforced via a **`conventions`** clause (i.e., a representation invariant). The representation conventions may be assumed to hold before and after each procedure implementation, except initialization, and must be confirmed to hold after every external procedure, except finalization. In this case we use it to assert that the `Front` and `Length` fields must fall within `Max_Length`.

The **`correspondence`** clause—sometimes also referred to as an abstraction function (or relation)—documents how to interpret the internal representation value of the queue as an abstract value. This relationship must be well founded. In particular, the **`correspondence`** must relate each legitimate representation value (i.e. those satisfying the **`conventions`**) to legitimate abstract values (i.e. those satisfying the **`constraints`** on the model). This is illustrated in Fig. 3.8.

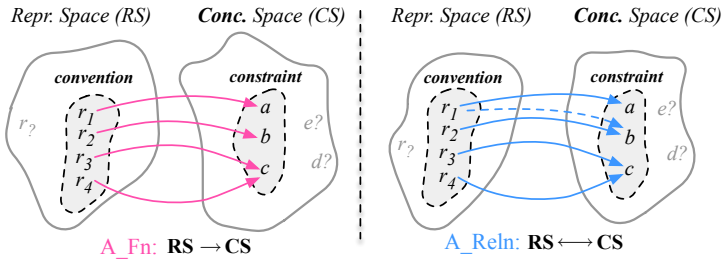


Fig. 3.8: Relationships between **`conventions`**, **`constraints`**, abstraction functions (left in pink), and relations (right in blue)

In the case of current example, the **correspondence** uses an iterated ("big product") string concatenation operator to assert that the contents, beginning with the Front index up-to the index one less than the queue's length (modulo Max_Length) is the same as the conceptual queue (i.e., Conc.Q). In a perhaps more 'canonical' mathematical notation we could express this as follows,

$$\texttt{Conc}.Q = \prod_{i=Q.Front}^{Q.Front+Q.Length-1} \langle Q.Contents(i \bmod Max\_Length)\rangle.$$

The implementation of Enqueue is made straightforward through the use of RESOLVE's standard integer and array operations. And since these operations are all formally specified, the RESOLVE compiler is able to transform the code shown into a collection of VCs which, if proven, establish the correctness of this particular realization. The VC badges appearing in Fig. 3.7 correspond to VCs that arise in the context of this realization.

VCs can arise from several places throughout the code. Some examples include: establishing that the realization as a whole satisfies the external interface specification (i.e., Queue_Template), showing that the code is consistent with internal assertions (e.g., the **conventions** clause), and showing that no violations occur in implicit and explicit calls on other objects (e.g., ensuring array accesses do not violate array bounds). We defer further discussion on the presentation and verification of VCs in the web-based IDE until Sect. 3.4.

Clean Semantics. The notion of clean semantics is the topic of [46, 61]. To summarize: in a language with clean semantics, only the values of objects that are explicitly touched are affected. For example, to avoid coupling an enqueued entry to the queue in which it is placed (through reference copying)—and subsequent indirect changes to one when the other is modified—the Enqueue operation has been specified using **alters** mode to avoid the need for copying altogether. As a result, an implementation can simply swap the enqueued entry into the array, thus avoiding any copying associated with assignment. This can be observed through the use of the swap operator :=: in the implementation of Enqueue:

```
Q.Contents[(Q.Front + Q.Length) mod Max_Length] :=: e
```

The statement shown swaps the value on the left with that on the right in constant time. This makes it possible to move arbitrarily large, complex structures (which is certainly plausible since `Entry` is a generic) efficiently without introducing aliasing. In the majority of use cases such an operator can obviate the need for otherwise intricate alias ownership management schemes.

3.4 Enhancements

RESOLVE also supports a form of specification inheritance via 'enhancement' modules. Such modules allow additional functionality to be layered on-top of pre-existing concepts. Our current component library includes a number of enhancements to `Queue_Template`, including the capability to append one queue to another (as shown in the component browser back in Fig. 3.3). Relationships between the queue concept, the append capability enhancement, and some general realizations are shown in the UML diagram below (Fig. 3.9).

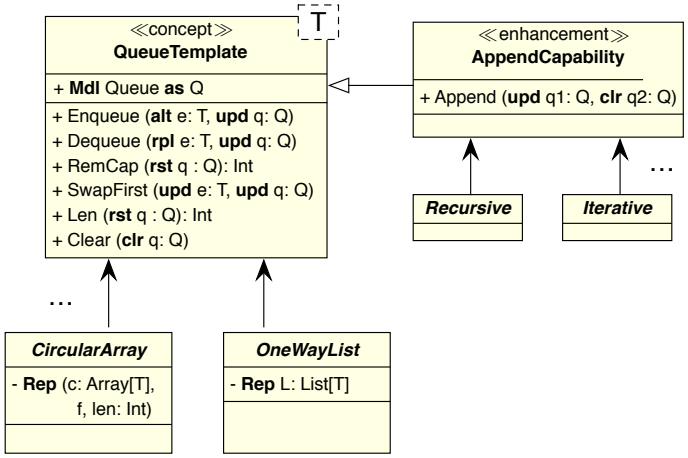


Fig. 3.9: A UML diagram of a queue enhancement and its realizations

The diagram communicates the public (+) and private (-) contents of each interface and its multiple (interchangeable) realizations—the titles of which we italicize. Here, a "realizes" relationship is communicated via solid lines with a filled-in pointy head, while lines with an empty arrowhead indicate an "extends" relationship between two interfaces. If the interface is parameterized by a generic type, we follow UML convention and place it in a white dashed box in the upper right hand corner [98].

Some Additional UML Conventions. To help keep UML diagrams in this work succinct, we make some concessions (beyond simply hiding the specifications) for presentation purposes:

- First, we precede each model type exported by an interface with the **Mdl** keyword, and allow its full name to be abbreviated (after the **as**) so as to shorten the proceeding operation signatures.
- Second, the names of operations are abbreviated (by removing underscores) along with the modes on their formal parameters. For example, we use `alt` for `alters`, `upd` for `updates`, `rst` for `restores`, `rpl` for `replaces`, `clr` for `clears`, etc.

3.4.1 An Enhancement for Appending Queues

Like the specifications for (primary) operations in a concept, 'secondary' operations in enhancement interfaces are also conceptual, and hence implementation neutral. The `Append_Capability` enhancement specification is shown below.

RESOLVE / Web IDE

```
Enhancement Append_Capability for Queue_Template;

  Operation Append (updates P : Queue, clears Q : Queue);
    requires |P| + |Q| <= Max_Length;
    ensures P = #P o #Q;

end Append_Capability;
```

RESOLVE / Web IDE

Here, the specification `requires` that the added length of both queues is within bounds and `ensures` (i) that the outgoing queue `P` is the concatenation of the two incoming queues (i.e., `#P o #Q`), and (ii) that the outgoing `Q` is cleared (as per the parameter mode on `Q`). Since the specification shown merely states (conceptually) what it means to append two queues, there are naturally multiple implementations one could write.

In Fig. 3.10 for example, we show an iterative realization of `Append_Capability` verified in the web-IDE.

The realization iteratively dequeues each entry from `Q` and re-enqueues the most recently dequeued entry onto the end of `P`. Note that the implementation relies only on the primary operations

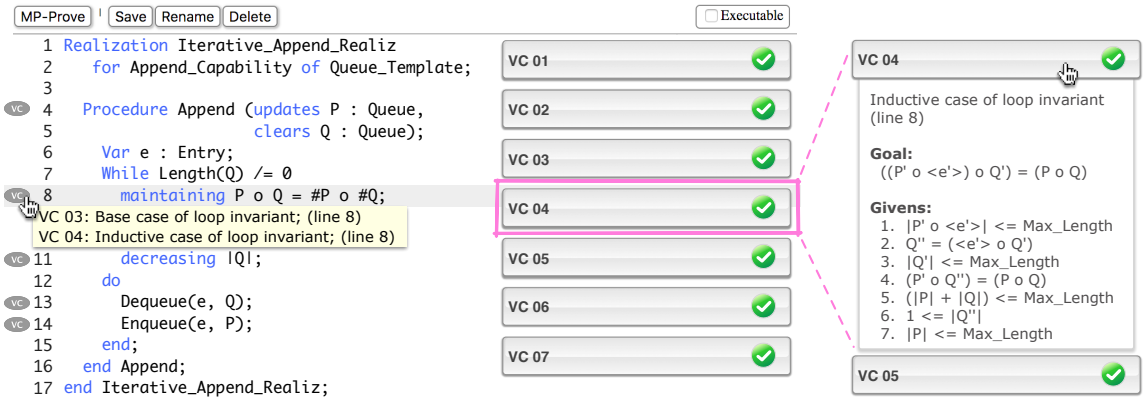


Fig. 3.10: An iterative realization of `Append` verified in the web-IDE

provided by `Queue_Template`. Indeed, as reflected in the UML diagram in Fig. 3.9, one of the major benefits of enhancements is the fact that they can be decoupled: that is, they can be written, implemented, and verified independently of any one particular realization of the base concept.

3.4.2 Verifying `Append`

Pressing the **MP-Prove** button (Fig. 3.10, upper left corner) invokes RESOLVE's automated verifier, which mechanically generates and attempts to prove automatically all of the VCs arising from the code. Recall that a VC badge in the left side gutter indicates the presence of one (or more) VCs. For example, on lines with a call statement, a VC is raised to verify that the pre-condition of the called operation holds. Hovering the cursor over one such badge reveals the specifics of the VC(s) generated.

On the right hand side of Fig. 3.10, the ✅ icon denotes a successfully proved obligation, while ⚠ is used for all other cases. It's uncommon for most programs to fully verify on the first attempt. Thus, to provide users with reasonably quick feedback, the prover's timeout has been capped at the relatively low value of 3000 milliseconds per each VC (at the risk of not being able to discharge some otherwise provable VC that might require more time).

To get a better sense of what a concrete VC looks like, and the role theories play in their verification, consider VC #4 from Fig. 3.10 presented in sequent form—where we label each antecedent (or, given) with a number $n \in \mathbb{N}^+$ to facilitate discussion of the proof:

$$\left.\begin{array}{l}(1)\ \texttt{|P' o <e'>|} <= \texttt{Max\_Length},\\ (2)\ \texttt{Q''} = \texttt{(<e'> o Q')},\\ \qquad\vdots\\ (4)\ \texttt{P' o Q''} = \texttt{P o Q},\\ (5)\ \texttt{|P|} + \texttt{|Q|} <= \texttt{Max\_Length}\end{array}\right\} \Gamma \quad \vdash \quad \underbrace{\texttt{(P'} \; o \; \texttt{<e'>) o Q'} = \texttt{P o Q}}_{\Delta}$$

This particular obligation corresponds to verifying the inductive case of the user-supplied loop invariant. Recall that Γ is the set of well formed formulas (wffs) that serve as antecedents/-givens while Δ in this case is a singleton set representing the succeedent/goal. Variables appearing with one or more primes (i.e., P', Q'', etc.) are intermediate (abstract) variables generated to reflect the values of the original (program) variables at different states in the code being verified.

Verification Steps.

The steps to verify VC #4 are relatively straightforward. To demonstrate, we first substitute the right hand side (rhs) of antecedent (2) for the occurrence of Q'' in antecedent (4) to obtain:

$$(4') \ \texttt{P' o (<e'> o Q')} = \texttt{P o Q} \quad \vdash \quad \texttt{(P' o <e'>) o Q'} = \texttt{P o Q}.$$

Next, we employ a corollary from Basic_String_Theory stating that the concatenation operator o is associative:

```
Corollary Cat_Assoc:
    Forall u, v, w : String, u o (v o w) = (u o v) o w;
```

Intuitively, having such a result present allows the automated verifier to *match* the left hand side (lhs) subterm of antecedent (4′) by 'instantiating' Cat_Assoc's bound variables *u*, *v*, and *w* as follows:

```
Forall u, v, w, (u o (v o w) = (u o v) o w)[u⇝P',v⇝<e'>,w⇝Q'].
```

Next, we simply substitute the matched lhs subterm in (4′) with the rhs of the instantiated equality in the body of Cat_Assoc resulting in,

$$(4'') \ \texttt{(P' o <e'>) o Q'} = \texttt{P o Q} \quad \vdash \quad \texttt{(P' o <e'>) o Q'} = \texttt{P o Q}$$

which the prover can automatically establish (since the same wff appears in both Γ and Δ).

3.5 Putting It Together

Clients can compose and use the component designed in this chapter with any chosen enhancements (such as append) by creating a new facility module, Fig. 3.11:

```
Facility Example_Fac;
    uses Basic_String_Theory;

    Facility QF is Queue_Template(Integer, 4)
        realized by Circular_Array_Realiz
    enhanced by Append_Capability
        realized by Iterative_Append_Realiz;

    Operation Main ();
    Procedure
        Var Q, P: Queue;
        Var x: Integer;

        --Create the queue <33, 44>
        x := 33; Enqueue(x, Q);
        x := 44; Enqueue(x, Q);

        --Create the queue <55, 66>
        x := 55; Enqueue(x, P);
        x := 66; Enqueue(x, P);

        --Append em'
        Append(Q, P);
```

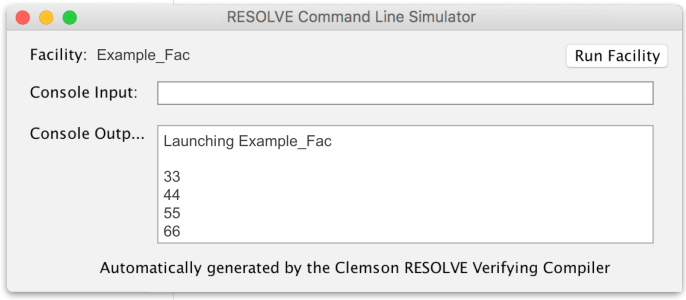


Fig. 3.11: Executing client facility code via a web-IDE generated `.jar` (note: print statements are omitted)

To use the queue and its enhancements, clients must instantiate a facility that pairs a specification of the desired component with a valid realization. Here, the facility `QF` produces queues of integers of length four and uses as its realization the circular array implementation given in Fig. 3.7. The **`enhanced by`** portion that follows pairs the desired extension (`Append_Capability`) with the iterative realization discussed and verified in Sect. 3.4.1. Users can layer any number of further enhancement-realization pairs onto a facility.

Chapter 4

The RESOLVE Specification and Proof System: Terms, Types, and Tools

This chapter introduces a number of topics—both theoretical and tool driven—that underpin the newest version of the RESOLVE compiler, and, by extension, the F-IDE and components we present in future chapters. Much of the formalization of the mathematics in this chapter, including the presentation of abstract syntax, accompanying definitions, and aspects of RESOLVE's math type system (including a recognition construct for user-defined subtyping) are new.

The chapter is organized into three parts. In part one we provide a high level overview of the underlying proof system and fix some notation—including abstract syntax for the newest version of the compiler. In part two, with the help of an illustrative example, we discuss RESOLVE's math type system for checking the well-formedness of specifications prior to verification. In part three we tie each of these aspects together with a proof of concept for a general purpose GUI-based compiler front-end termed "VerifierGUI" (or, VGui for short). We conclude the chapter with a demonstration of the tool by using it to derive and prove verification conditions (VCs) arising from code involving a set-like data structure called the search store template.

Lastly, on a typographical note, we use a san-serif style font when typesetting the names of any built-in inference rules, meta-operators, or types that we introduce along the way.

4.1 A Proof System for RESOLVE: Basic Concepts and Notation

The purpose of this section is to fix notation for a formal discussion of RESOLVE's sound and (relatively) complete proof system, and to set the stage for a more thorough demonstration of the VC derivation process as well as automated verification in the context of the VerifierGUI tool (Sect. 4.3).

And though this section is not a comprehensive treatment of the proof system or its rules (consult [48, 101] for this), we do however review the general process of transforming programs into formal assertions and establish abstract syntax for RESOLVE's integrated specification language.

4.1.1 Assertive Code

Generation of VCs that are both necessary and sufficient in order to prove that an implementation is correct w.r.t. its specification is a syntax-directed process. Specifically, each language construct (i.e., statements, declarations, etc.) has an associated inference rule that dictates how to transform that particular construct into a mathematical assertion.

Prior to the application of any statement-level proof rules however is a pre-processing step in which user code is logically grouped into *assertive-code* blocks wherein all mathematical assertions are made explicit. The traditional Hoare triple of the form $\{P\}c\{Q\}$ is expressed as follows in our notation:

$$\mathcal{C} \setminus c; \quad \texttt{Confirm}\ Q.$$

Here, $\mathcal{C}$ is the context containing a collection of typed symbols obtained from declarations encountered when processing one or more modules, c is a sequence of zero or more program statements (interleaved with specification statements such as `Assume` P), and Q is an assertion that must be confirmed to hold at the end.

Before discussing the rules that operate on assertive code blocks, we first establish abstract syntax and define some other fundamental operators in the following section.

4.1.2 Abstract Syntax

Let $\mathcal{T}$ be a set of types initially containing $\{\mathsf{HB},\ \mathsf{Set}\} \in \mathcal{T}$, where HB denotes the (meta) type **H**yper **B**oolean for booleans[1] and Set represents a hyper set type from which we can create arbitrary new types $T : \mathsf{Set}$. See Sect. 4.2 for more on types. The specification language itself is organized into several constituent parts:

- A set $\mathcal{X}$ of typed terminal symbols $s, x : T \in \mathcal{X}$ where s, x could also have any type in $\mathcal{T}$.
- A set $\mathcal{F}$ of typed function symbols $f, g, .., h : T_1 \times .. \times T_n \longrightarrow T$ which range over $\mathcal{F}$. Here $T_1 \times .. \times T_n$ represents the domain, while the (non-subscripted) T represents the codomain. We denote the arity of a given function symbol f as $\mathsf{ar}(f)$. Thus, $\forall s \in \mathcal{X}, \mathsf{ar}(s) = 0$.
- Lastly, a set $\mathcal{P}$ of typed predicate symbols $P : T_1 \times .. \times T_n \longrightarrow \mathsf{HB}$; which also includes a reserved binary predicate for equality ($= : \mathsf{Set} \times \mathsf{Set} \longrightarrow \mathsf{HB}$) as well as nullary predicate symbols for true and false.

These sets, when combined, constitute the language's vocabulary $\mathcal{V} = \mathcal{X} \cup \mathcal{F} \cup \mathcal{P}$; where $\mathcal{V}$ can be enriched via the definition of new constants, functions, and predicates in mathematical theories.

Theory Modules and Definitions. The normal mechanism for adding symbols from any one of these categories to some context $\mathcal{C}$ during compilation is via theory modules which contain one or more definitions and/or theorems. The general essence of a theory module `T` is shown show.

```
Precis T (extends T_P)? //the ? indicates the extension is optional
  uses Th_1, ... ,Th_n;

  Def f : ℤ ⟶ ℤ;
  Literal Def 0 : ℤ;
  Theorem f_Inj: ∀ x, y : ℤ, f(x) = f(y) ⟹ x = y;

  definitions, theorems, and corollaries ...
end T;
```

[1] "Hyper" types, which we discuss in Sect. 4.2.1, are part of a general framework being developed to describe the mathematics of RESOLVE from the outside

Here, we use definitions (declared via the `Def` keyword) to introduce an uninterpreted function `f` into scope along with another integer-typed symbol `0`. The `Literal` modifier preceding the second definition tells the compiler that the declared symbol is to be interpreted internally as a constant/literal. While usage of this modifier is not strictly necessary, it does allow the compiler to perform certain simplifications automatically as part of the verification condition derivation process. Precis modules can also extend another (base) precis. This merely allows one to group related operators or theorems together that are perhaps not general enough to warrant inclusion in the parent precis. Each precis is supported by a separate module containing proofs of the various theorems and corollaries that appear in a given precis. While proof modules are an integral part of RESOLVE [97], this dissertation is not directly concerned with them. As such, the F-IDE we present in later chapters does not yet include support for their development.

We now fix a grammar for RESOLVE formulas (which denote truth values) and terms (which serve as the fundamental building blocks of formulas). We occasionally use the general descriptor "term" or "expression" to mean either one of these.

Definition 1. *The set of formulas and terms of our specification language over vocabulary $\mathcal{V}$ is given by the following abstract syntax.*

$$\begin{aligned}
&\textbf{Form}_{\mathcal{V}} \ni \phi, \psi ::= P(t_1, .., t_{\mathsf{ar}(P)}) \mid \mathsf{true} \mid \mathsf{false} \mid \neg\phi \mid \phi \circ \psi \mid \mathcal{Q}\bar{x}_n, \phi \mid t \\
&\textbf{Term}_{\mathcal{V}} \ni \tau, t, y ::= t_0(t_1, .., t_{\mathsf{ar}(t_0)}) \mid t \longrightarrow y \mid t : \tau \mid t_0 \times .. \times t_n \mid t.y \mid \lambda \bar{x}_n, t \\
&\qquad \mid \left(\begin{array}{l} \mid\{\ t \quad \texttt{if } \psi \\ \mid\{\ y \quad \texttt{otherwise} \end{array} \right)^{+} \\
&\qquad \mid \{\bar{x}_n \mid \psi\} \mid (\psi) \mid \#?\ s
\end{aligned}$$

where $\circ \in \{\wedge, \vee, \Longrightarrow, \Longleftrightarrow\}$, $\mathcal{Q} \in \{\forall, \exists\}$ and $\bar{x}_n$ is shorthand for a non-empty list of typed binder variables: $x_1, .., x_n : \tau$.

Specifically, formulas consist of the usual logical connectives and quantifiers while terms permit (reading from left to right) function application,[2] function (type) constructors, embedded type inhabitation assertions ($t : \tau$), cartesian products ($\times$) and field selectors ($t.y$), lambda

[2] Outfix and infix style applications are also accepted, though we omit these for brevity

abstraction, conditional definitions, set comprehensions, parenthesized formulae, and—finally—potentially incoming (#-denoted) terminal symbols s. We reserve additional remarks on these syntactic constructs for when they appear in future examples.

Lexically, the name of a terminal symbol s in a mathematical assertion can take a variety of forms—expressed via the following lexical charsets:

- **Unicode Glyphs.** A unicode glyph can range over a collection of arrow-like symbols ($\Longrightarrow$, $\longleftarrow$, ..,$\Longrightarrow$), double struck and other calligraphic letter-like characters ($\wp$, $\mathbb{N}$, $\mathbb{Z}$, ..,$\mathbb{Q}$), typical operators or relations ($\odot$, $\ltimes$, $\cup$, $\equiv$, .., $\leq$), 'big operators' ($\bigcup$, .., $\bigvee$), and, lastly, Greek characters (α, β, .., Γ).

- **Subscripts, Superscripts, and Primes.** Any mathematical identifier can also be proceeded by zero or more alphanumeric unicode subscript or superscript characters followed by zero or more prime characters.

- **ASCII and Identifiers.** Comprises the usual keyboard symbols including (`+`, `-`, `*`, `>`, `<`, `=`, `~`) as well as regular (program) identifiers.

These categories can be combined to form fairly intricate names such as $\mathbb{N}$`+`, `=<>=`, Γ_0, `f'`, `f''`, etc.

Next, we establish our syntax for statements and assertive code fragments.

Definition 2. *A fragment of assertive code consists of programmatic statements interleaved with assertive statements of the form:*

$$\textbf{Stmt}_{\mathcal{V}} \ni s ::= \texttt{Assume}\ \phi; \mid \texttt{Confirm}\ \phi; \mid \texttt{Stipulate}\ \phi; \mid id :=: id; \mid \cdots \mid id(y_1,..,y_n);$$
$$\textbf{AsrtCode}_{\mathcal{V}} \ni a ::= s^*\ \ \texttt{Confirm}\ \bigwedge seq^+; \qquad \textbf{Seqnt}_{\mathcal{V}} \ni seq ::= \Gamma \vdash \Delta$$

where Γ *and* Δ *are sets of well-formed-formulas (wffs).*

The $\textbf{Stmt}_{\mathcal{V}}$ production rule admits assertional verification language statements (including `Assume`, `Confirm`, and `Stipulate` clauses—which we elaborate on further in Sect. 4.3.3) as well as strictly programmatic ones such as swap ($id :=: id$) and procedure calls: $id(y_1,..,y_n)$. We omit syntax for any remaining programmatic statements for brevity. Assertive code, ($\textbf{AsrtCode}_{\mathcal{V}}$) in turn,

consists of zero or more statements followed by a special "final confirm," which is a conjunction of one or more sequents (where each sequent is expressed via the $\mathbf{Seqnt}_{\mathcal{V}}$ production).

Lastly, since some of the proof rules we present in future sections require the ability to extract programmatic "free variables" from a confirmed or assumed assertion, we define a "assertion free variable" operator $\mathsf{AFV} : \mathbf{Form}_{\mathcal{V}} \longrightarrow \wp(\mathbf{Term}_{\mathcal{V}})$.

Definition 3. *The set of assertional free variables of some formula ϕ appearing in syntactic context $\mathcal{C}$—e.g.: $\mathcal{C}\backslash$*`Assume` $\mathsf{AFV}(\phi)$*—is defined as follows:*

$$\mathsf{AFV}(t_0 \odot t_1) = \mathsf{AFV}(t_0) \cup \mathsf{AFV}(t_1) \text{ where } \odot \in \{\wedge, \vee, \Longrightarrow, \Longleftrightarrow\}$$

$$\mathsf{AFV}(t_0(t_1, .., t_{\mathsf{ar}(t_0)})) = \bigcup_{i=1}^{\mathsf{ar}(t_0)} \mathsf{AFV}(t_i) \cup \begin{cases} \{t_0\} \textbf{ if } \mathcal{C}\backslash t_0 \text{ has designation } \textbf{PVar} \text{ (prog. var)} \\ \mathsf{AFV}(t_0) \textbf{ if } \neg \mathsf{IsSyntacticVar}(t_0) \\ \emptyset \textbf{ otherwise} \end{cases}$$

$$\mathsf{AFV}(s) = \begin{cases} \{s\} \textbf{ if } \mathcal{C}\backslash s \text{ has designation } \textbf{PVar} \\ \emptyset \textbf{ otherwise} \end{cases}$$

$$\mathsf{AFV}(\mathcal{Q}\, \bar{x}_n, t) = \mathsf{AFV}(t) - \{x_1, .., x_n\}$$

Here, the meta predicate $\mathsf{IsSyntacticVar} : \mathbf{Term}_{\mathcal{V}} \longrightarrow \{\mathsf{true}, \mathsf{false}\}$ holds iff the term it receives is (1) a possibly #-preceded terminal symbol, or (2) a segmented (product) 'selector term' $s_0.s_1.\ \cdots\ .s_n$ where $\mathsf{IsSyntacticVar}(s_i)$ holds.

The AFV definition shown differs from the (synthesized) free variable function typically presented in logic textbooks as it extracts only variables with a "programmatic variable" designation called **PVar**. Whether or not a variable has this designation is determined by the source of its declaration and whether or not it carries an associated programmatic type—thus making the operator strongly dependent on the context $\mathcal{C}$. The VC generator uses these extracted variables to determine whether a particular formula should be added to an existing sequent within a fragment of assertive code. Though used in Sect. 4.3.3, we first illustrate the definition with an example.

Example 1. Suppose the context $\mathcal{C}$ initially contains the symbols:

$$\mathcal{C} = \{\ \texttt{Entry}_p : \mathsf{SSet},\ = : \mathsf{Set} \times \mathsf{Set} \longrightarrow \mathsf{HB},\ \cdots\ \}.$$

Here, the subscripted p on `Entry` indicates that it originated as a generic (programming) type parameterizing a concept. Next, we enrich $\mathcal{C}$ with a made-up operation `M` that **`updates`** its parameter `A` of type `EntryMap` (which we assume is **`modeled by`** `Entry` $\longrightarrow$ `Entry`).

$\mathcal{C}' = \mathcal{C} \cup \{$**`Oper`** `M (`**`preserves`** `e : Entry;` **`alters`** `x : Entry,`
`updates A : EntryMap);`
`requires` `x` $\neq$ `e;`
`ensures` ($\forall$ `y : Entry, y` $\neq$ `e` $\implies$ `A(y) = e)` $\wedge$
`A(e) = #x` $\wedge$ `Img(A)` $\neq$ `Img(#A);`

If we denote the formula in the **`ensures`** clause above as ψ, then applying $\mathcal{C}' \backslash \mathsf{AFV}(\psi)$ yields: {`#A`, `A`, `e`, `#x`}. Note that purely mathematical operators used in applications (such as `Img`, $\wedge$, $\leq$, `=`, etc) are excluded, whereas the 'named-subterm' `A` of an application such as `A(x)` is included as it carries an implicit **PVar** designation due to its origin (and typing) as a formal program parameter. Note too that that program-typed variable `y` is excluded, as it's bound under the quantifier.

Handling Term and Sequent Equality. In this work we handle term equality by saying two terms t_1 and t_2 are "alpha equivalent" (i.e. $t_1 \equiv_\alpha t_2$) iff both terms share exactly the same structure—and differ only in the names of any bound variables occurring therein (so long as the names retain the original semantic meaning of the term). For example,

$$\lambda x : \mathbb{Z}, \lambda y : \mathbb{Z}, \varphi(x, y, v) \quad \equiv_\alpha \quad \lambda i : \mathbb{Z}, \lambda j : \mathbb{Z}, \varphi(i, j, v)$$

since these terms share the same fundamental relationship between the bound variables appearing in the inner term. However,

$$\lambda x : \mathbb{Z}, \lambda y : \mathbb{Z}, \varphi(x, y, v) \quad \not\equiv_\alpha \quad \lambda i : \mathbb{Z}, \lambda i : \mathbb{Z}, \varphi(i, i, v)$$

since renaming the bound variable of the inner lambda (on the right hand side) to i inadvertently captures the first argument to φ—which was originally bound in the outermost lambda.

This can be efficiently solved using a so called "locally nameless representation" which replaces the names of bound variables with indices—though the implementation of this non-trivial in general [40]. So to avoid additional complexity, this work employs a less efficient (but com-

paratively straightforward) naive implementation based on repeated substitution and bound variable renaming.

Syntactic Sequent Equality: This notion of equality extends to sequents as well. Specifically we say two sequents $\mathbf{S}_1$ and $\mathbf{S}_2$ are alpha equivalent (i.e. $\mathbf{S}_1 \equiv_{\vdash\alpha} \mathbf{S}_2$) iff the antecedent and succedent share the same cardinality and contain syntactically equivalent formulae (as determined by $\equiv_\alpha$ on terms).

4.1.3 Verification Process

Once assertive code has been constructed, the approach we use to generate VCs is *goal-directed* and is illustrated at a high level in Fig. 4.1.

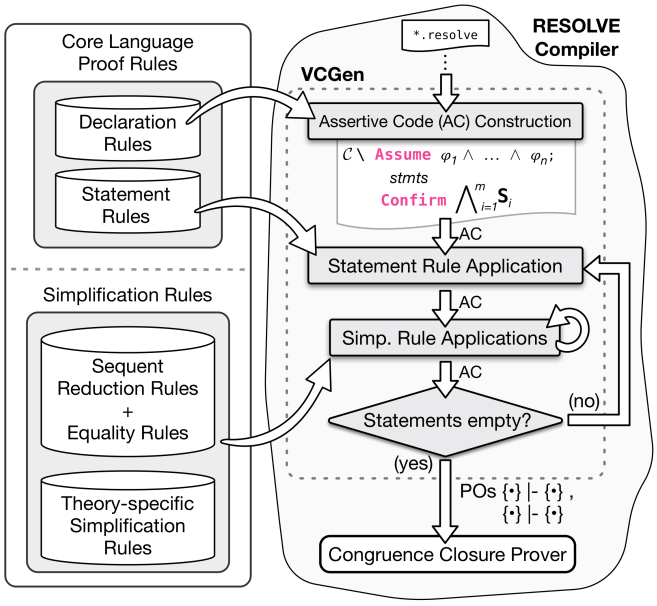


Fig. 4.1: An overview of RESOLVE's goal directed verification condition generation scheme along with the various categories of rule types

Starting with the penultimate statement (immediately prior to the conjunction of sequents $\mathbf{S}_1 \wedge .. \wedge \mathbf{S}_m$ that always terminates an assertive code block), statements are eliminated one at a time via the application of their corresponding proof rules—where each application effects one or more sequents (and their wffs) in the final `Confirm`. After each statement rule application, we apply a round of the sequent reduction rules shown in Fig. 4.2 to the final `Confirm` assertion to eliminate any

introduced logical connectives.[3] Theory-specific rewrite rules, which we have developed suitable machinery to support—though ultimately leave as future work—could also be applied at this point as well.

Fig. 4.2: Standard sequent reduction rules for logical connectives $\wedge$, $\vee$, $\Longrightarrow$, and $\neg$; where ϕ and ψ denote arbitrary formulae

$$\textsf{AndLeft}\ \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta} \qquad \textsf{AndRight}\ \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}$$

$$\textsf{OrLeft}\ \frac{\Gamma, \phi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \vee \psi \vdash \Delta} \qquad \textsf{OrRight}\ \frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta}$$

$$\textsf{ImpLeft}\ \frac{\Gamma \vdash \phi, \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \Longrightarrow \psi \vdash \Delta}$$

$$\textsf{ImpRight}\ \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \Longrightarrow \psi, \Delta}$$

$$\textsf{NotLeft}\ \frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta} \qquad \textsf{NotRight}\ \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}$$

After each statement is eliminated in a given block of assertive code, the conjuncted sequents left over in each branch are broken apart and become the final VCs that are ultimately sent off to RESOLVE's in-house congruence closure prover for verification. Note however that the VC derivation process described may result in multiple "branches" of assertive code (e.g., after applying the IfThenElse rule, assertive code splits into two cases where one assumes the conditional was true while the other assumes it was false and skips the code in the body accordingly [101]). The system thus takes measures to avoid unnecessary duplication of sequents across separate branches of assertive code through a combination of caching, location information internally associated with each sequent, and the term/formula/sequent equality tests discussed in the previous section.

Each proof rule, including theory specific ones can be formally expressed in the form:

$$\textsf{ruleName}\ \frac{H_1 \ \cdots \ H_n}{C}$$

where H_i is called the premisses and C the conclusion of the rule. To illustrate how the rules appearing in Fig. 4.2 are applied to a concrete example, we consider the following sequent:

[3] see [16] for a fairly recent presentation of these (and other) sequent rules

$$\vdash p \wedge q \Longrightarrow q \wedge p$$

Since the rules can only be applied to top level formulae (indicated by the presence of the turnstile $\vdash$ in each), our only available option is to apply ImpRight, which results in: $p \wedge q \vdash q \wedge p$. Applying AndLeft to this produces $p, q \vdash q \wedge p$ then applying AndRight splits the sequent into two: $p, q \vdash q$ and $p, q \vdash p$—which are both trivial since the same wff appears on either side of the turnstile in each. We demonstrate these ideas in action later in Sect. 4.3.3.

4.2 Mathematical Type Checking in RESOLVE: An Overview

There are a number of reasons why a type system for checking formal theory developments and specifications is a worthwhile objective in the overall design of a verifying compiler.

Benefits range from eliminating clear errors in theories and specifications, to making the system more usable and responsive in practice by providing compile time, type level feedback to specifiers. Such feedback is of particular importance, as any user who interacts with RESOLVE's specification language will necessarily need to think about and satisfy typing constraints—which, in turn, influences features and feedback present in front end tools (including the F-IDE we present in the proceeding chapter).

Another major benefit of well-formed, type-checked specifications is that it has the potential to simplify the design of the automated prover by: (1) eliminating the need to interpret malformed verification conditions and (2) using type information to guide the prover towards relevant theorems and corollaries in a precis. As a general design principle then, we seek to keep the concerns of the prover and its associated logic separated from the (comparatively 'easier') concerns of the mathematical type checker.

This principle of separation however is not always a given. In particular, systems based on intuitionistic type theory such as Agda [84] and Coq [9] do not differentiate between type-checking and proving. Rather, under such systems, arbitrary assertions (including those quantifying over sets of values) are encoded as types. Thus, in order to prove some formula F expressed as a type, users

must construct a program that can be shown to have type F.

Though powerful, such an approach has some notable downsides. Not only does it (unsurprisingly) make type checking undecidable in general, but it also places a fairly large burden on users. Indeed, one must become comfortable with functional programming paradigms, the intricacies associated with the typechecker and its re-rewriting/tactic engine, as well as comfortable in thinking about types as proofs.

Example: Proving a Simple Theorem in Agda. To illustrate the approach to proving such systems take, consider the following Agda theorem about the `not` operator:

```
nn-elim : forall (b : Bool) -> not not b ≡ b
nn-elim true  = refl
nn-elim false = refl
```

Here, the type of `nn-elim` (after the `:`) is a "universal type" that takes an arbitrary boolean `b` and produces (after the `->`) a propositional equality type: `not not b ≡ b`. The two defining equations that follow cover both possible inputs to `nn-elim`. In the first case, the theorem is instantiated with the value `true` and the "output type" becomes `not not true ≡ true`. At this point, Agda's simplifier (using the definition of `not`) will automatically perform the rewrites necessary to attain `true ≡ true` (which is proven via `refl`exivity).

In RESOLVE we opt for a balanced approach that on one hand is less expressive (as we disallow arbitrary propositions as types—such as "universal types"), yet still capable of leveraging generic definitions and higher-order functions that range over arbitrary classes of functions, sets, and powersets/powerclasses.

This section is by no means intended to be an exhaustive or complete description of RESOLVE's work-in-progress mathematical typechecking system. In particular, our treatment of user-defined subtyping and instantiation of generic type signatures merely represent starting points. Nevertheless, the type-system we discuss in this section is developed to a point where it provides useful feedback for supplementing error reporting in tools (IDEs or otherwise).

4.2.1 Type System Organization In a Nutshell

The intent of this section is to provide a big picture overview of RESOLVE's math type system, the organization of which is depicted in Fig. 4.3.

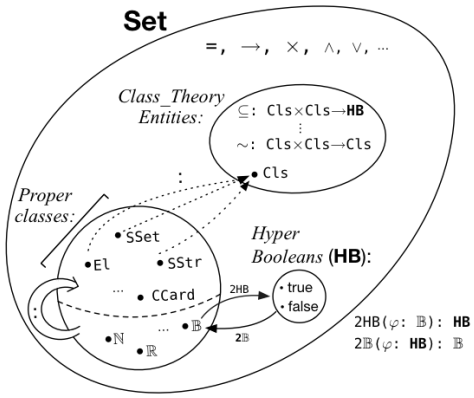


Fig. 4.3: An (abridged) overview of RESOLVE's math type universe

The hyper (or, meta) set theory we use to describe the mathematics of RESOLVE is represented by the outermost circle. This should be interpreted as a first order theory which includes a few built-in sorts (or, collections/types) including a universal sort **Set** as well as a sort for **HB** for booleans. Thus, each logical operator introduced in the $\mathbf{Form}_{\mathcal{V}}$ rule ($\wedge, \vee$, etc., see Def. 1) inhabits this space.

This first order framework, in turn, is used to describe another meta foundational theory called "class theory," which is a variant of set theory that operates over large collections termed proper classes: or, `Cls`.[4] The general idea is to use such classes to represent collections of objects such as the class of all mathematical strings (`SStr`) or the class of cardinal numbers (`CCard`). The class of all sets is `SSet`, and some members of this are shown below the dashed line: e.g., $\mathbb{N}$: `SSet`.

Some General Points.

- Here, the type inhabitation relation is ':' and is built-in syntactically (but can be described at the meta level within class theory).

[4]The notion of proper classes appears in several common axiomatizations of set theory including (informally) in Zermelo-Fraenkel (ZFC), and more formally in von Neumann-Bernays-Gödel (NBG)—see, e.g., [74]

- The set-based booleans $\mathbb{B}$ can be modeled within this framework; however, since the logical operators by default produce a hyper boolean **HB**, specifications generally employ some composition of the the two. So the system automatically applies the translation operators $2\mathsf{HB}$ and $2\mathbb{B}$ to convert between the two.

- Since most type checking takes place at the site of function applications, the typing rule for this follows the expected convention: if `f` is a function with type signature `T` $\longrightarrow$ `U` and `t` is a term of type `T`, then `f(t)` is a term of type `U`. Note that, in general, the arrow operator $\longrightarrow$ associates to the right: thus `A` $\longrightarrow$ `B` $\longrightarrow$ `C` means `A` $\longrightarrow$ (`B` $\longrightarrow$ `C`). Partial applications of functions are also supported, though the signature must be expressed in a curried form (i.e., without the use of $\times$ in the domain).

4.2.2 Example: Formalizing a Set Application Functional

Rather than discussing each aspect of the type system in isolation, we simply consider an illustrative example that exercises some combination of its features. In particular, we define a set application functional (shown below) that will prove useful for the specifications in the proceeding section.

RESOLVE

```
Precis Set_App_Op_Ext extends Function_Theory;
  Def App (f : (T : SSet) ⟶ (U : SSet), S : ℘(T)) : ℘(U) ≜
        {u : U | ∃ x : S, f(x) = u};
  // ... see appendix A.4.1 for additional corollaries
end Set_App_Op_Ext;
```

RESOLVE

Based on its signature, the (higher-order) definition `App` takes a function `f` : `T` $\longrightarrow$ `U`, a set in `f`'s domain (`S`), and produces a set from the codomain of `f` by applying `f` on every point in `S`.

Before we consider `App`'s body and define some corollaries for this operator, we first draw attention to some of the more subtle aspects of its type signature.

Embedded Type Arguments. First, note that the definition uses the type assertion rule from Def. 1 ($t : y$) to embed the types for the domain (`T`) and codomain (`U`) within the function type assigned to `f`. The following example demonstrates how this provides the system with a flexible form of type inference.

Example 2. Suppose we have a simple theory that contains the following definitions (where `+` : $\mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$ is defined in `Integer_Theory`):

`Def Add_Two (i : ℤ) : ℤ ≜ i + 2;`

`Def M : ℘(ℤ) ≜ {1, 3, 0, 5};`

With these, we can apply the `Add_Two` function over `M` as follows:

`App(Add_Two, M) = {3, 5, 2, 7};`

Notice that it was not necessary to explicitly pass the the instantiations of `T` and `U` to `App` (as in, `App(ℤ, ℤ, Add_Two, M)`). Rather, the type system will automatically attempt to 'bind' the types of the actual arguments to the structure of their expected types—propagating any instantiated type variables (indicated by single quotes) across the signature and body once discovered.[5] For example,

`App: ('T' : SSet) ⟶ ('U' : SSet) × ('S' : ℘('T')) ⟶ ℘('U')`

after type instantiation for the application `App(Add_Two, M)` becomes:

`App: (ℤ ⟶ ℤ) × ('S' : ℘(ℤ)) ⟶ ℘(ℤ)`

at which point the type checker simply ensures that the types for actual arguments (i.e. `Add_Two`, `M`) are conformal with their expected types (in this case, they match exactly—though we later consider a situation where things don't work out so nicely).

Turning to the comprehension in the body, we note that (from `Class_Theory`) the set comprehension operator adheres to the following (meta) typing rule involving the powerset operator over arbitrary formulae:

$$\forall \psi : \mathbf{Form}_{\mathcal{V}}, \forall T : \mathtt{SSet}, \{x : T \mid \psi\} : \wp(T).$$

[5] This step also ensures that all instantiations are type conformal with the upper bounds imposed by the signature (in this case, `SSet` for `T` and `U`)

The rule states that any comprehension term that ranges over some type T can be considered a term of type $\wp(T)$. Such a rule in this case is needed as the body type of any functional definition (in this case, App) must be conformal with the result type specified in the function's signature. This rule suggests a general mechanism for defining new type level rules on terms, which we discuss next.

Recognition of Type Conformal Terms. Studying the body of App (after the $\triangleq$) two useful corollaries come to mind, which we state below:

— RESOLVE

```
Corollary App_C1:
    ∀ D, R : SSet, ∀ f : D ⟶ R, App(f, ∅) = ∅;
Corollary App_C2:
    ∀ D, R : SSet, ∀ f : D ⟶ R, App(f, D) = Im(f);
```

RESOLVE —

The first corollary App_C1 merely establishes that the application of any function f : D ⟶ R over the empty set always results in the empty set, while the second corollary App_C2 states that application of f over its domain D produces the image of f (this is also expressed—though perhaps less explicitly—by the comprehension in the body of App).

It's only however when we compile this theory that we run into a problem:

— Compiler Output

```
error(203): Set_App_Op_Ext.resolve:10: no function applicable with domain:
    (‘D’ ⟶ ‘R’ × Cls)
candidates:
  App : ((f : ‘T’ ⟶ ‘U’) × (t : ℘(‘T’))) ⟶ ℘(‘U’)
error(203): Set_App_Op_Ext.resolve:12: no function applicable with domain:
    (‘D’ ⟶ ‘R’ × SSet)
candidates:
  App : ((f : ‘T’ ⟶ ‘U’) × (t : ℘(‘T’))) ⟶ ℘(‘U’)
```

Compiler Output —

These errors indicate a problem with the application on the left hand side of the equality in each corollary. To help determine a suitable fix, we consider the instantiated signature for App:

```
(‘D’ ⟶ ‘R’) × (‘S’ : ℘(‘D’)) ⟶ ℘(‘R’)
```

In both cases, passing the function `f` into `App` correctly propagates `f`'s domain (`D`) and range (`R`) across the signature—the issue however lies with type checking the second argument:

- In the first case, `App(f, ∅)`, the system was unable to statically *recognize* a property of the $\wp$ operator: namely, that $\emptyset$ (of type `Cls`) is suitable where type $\wp(\text{`D'})$ is expected.
- The second case, `App(f, D)` is similar, only here the system failed to recognize that the argument `'D'` of type `SSet` is a member of $\wp(\text{`D'})$.

Indeed, since mathematical entities can potentially have *many valid type assignments*—and our system permits users to freely define new entities at will—this suggests the need for a construct that allows the type checker to recognize that an object belongs to some class (and, more generally, that values of a particular term belong to some class).

In this work we solve this problem using a new construct referred to as a 'recognition,' which adheres to the following rule schema:

$$\texttt{Recognition}\ (id)?: \forall x:\tau_1, \forall y:\tau_2, \ldots,$$
$$(\texttt{if}\ condTrm\langle x,y,..\rangle\ \texttt{then})?\ patternTrm\langle x,y,..\rangle : \tau\langle x,y\rangle.$$

Here, *patternTrm* identifies the concrete value or object we're attempting to identify as belonging to (type) term τ while *condTrm* specifies an (optional) guard condition that must be satisfied before the recognition can be used. Here we use the notation $t\langle\xi_1,..,\xi_n\rangle$ to identify a term t involving syntactic variables $\xi_1,..,\xi_n$.

Using this construct we can specify the following two recognitions—which allows corollaries `App_C1` and `App_C2` to typecheck.[6]

RESOLVE

```
Recognition Empty_Set_In_All_Powersets:
        ∀ S : SSet, ∅ : ℘(S);
Recognition Powerset_Mem:
   ∀ S : SSet, S : ℘(S);
```

RESOLVE

[6]These particular (fundamental) recognitions were already present in compiler's core `Class_Theory` module. We had simply commented them out to trigger the type checking errors motivating our discussion

This construct naturally is used throughout our theories to specify more common (type) membership properties as well. For example, consider the following recognition provided in an extension to natural number theory: `Recognition All_Nats_In_Z:` $\forall$ `n :` $\mathbb{N}$`, n :` $\mathbb{Z}$ which tells the compiler to allow a value of type $\mathbb{N}$ to be passed anywhere a term of type $\mathbb{Z}$ is expected.

4.3 From Code to Verification Conditions: Interactive Derivation Tracing and Simplification

In this section we discuss the RESOLVE proof system in the context of the VerifierGui (VGui) tool, which we've developed as a standalone, general-purpose means of viewing VCs and interacting with the compiler and its in house prover. Fig. 4.4 shows the VGui splash screen and a preview of the tool in use.

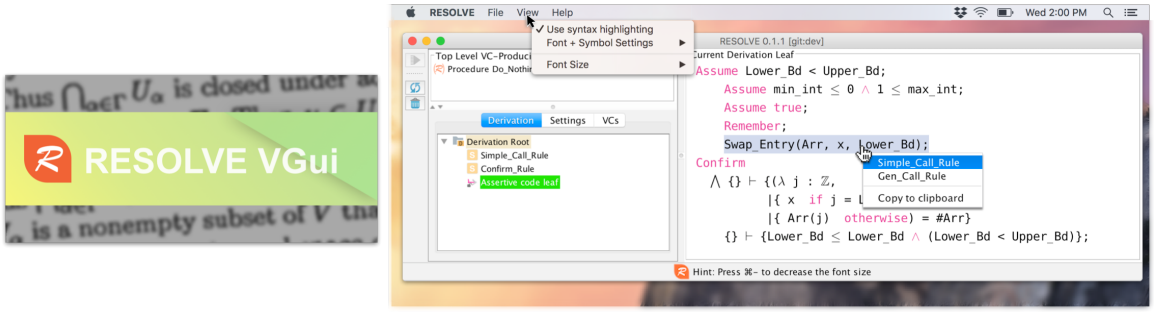


Fig. 4.4: The VerifierGui compiler front-end

In particular we discuss: (i) how the tool can be used to interactively and automatically derive proof obligations—offering advanced users/researchers more insight and control over how the system processes their specifications and code into VCs and (ii) how the tool can be generally used as a front-end for invoking RESOLVE's automated prover.

4.3.1 A Search Store Template Specification

The running example we use in this section is an enhancement to an unordered (set-like) data structure termed the `Search_Store_Template`. A snippet of this concept is provided below.

RESOLVE

```
Concept Search_Store_Template (type Key;
                               evaluates Max_Capacity : Integer):
  uses Class_Theory with Std_Cardinality_Ext, Basic_Natural_Number_Theory;
  requires 1 ≤ Max_Capacity which_entails Max_Capacity : ℕ;

  Type family Store is modeled by ℘(Key);
    exemplar S;
    constraints ‖S‖ ≤ Max_Capacity ...;
    initialization
          ensures S = ∅;

  Operation Add (restores k : Key; updates S : Store);
    requires ‖S‖ + 1 ≤ Max_Capacity ∧ k ∉ S;
    ensures S = #S ∪ {k};

  //note that we use ∼ to denote set subtraction
  Operation Remove_Any (replaces k : Key; updates S : Store);
    requires 1 ≤ ‖S‖
    ensures k ∈ #S ∧ S = #S ∼ {k};

  //remaining operations omitted for brevity
end Search_Store_Template;
```

RESOLVE

Much of the interface and its formalization relies on operators defined and exported through RESOLVE's foundational class theory (see Sect. 4.2.1). However, since class theory is not necessarily restricted to describing *countable* sets, the **uses** extension following the **with** clause adds necessary theory results involving the class of cardinal numbers `CCard` as well as the cardinality operator $\|\bullet\| : \texttt{Cls} \longrightarrow \texttt{CCard}$.

The type model for `Store` is defined to range over subsets of the powerset of the generic `Key` type passed into the module—with the **constraint** that the cardinality must fall within `Max_Capacity`. Note that since all programming types in RESOLVE (including generics like `Key`) inhabit `SSet`, we can apply `Key` to the powerset operator $\wp : \texttt{SSet} \longrightarrow \texttt{SSet}$ to obtain the desired mathematical model.

Additionally, to facilitate the type checking of assertions that mix cardinal numbers with natural number operators (e.g., $\|\texttt{S}\| \leq \texttt{Max\_Depth}$, where $\leq : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{B}$) we amend the

constraints clause as follows:

constraints ‖S‖ ≤ Max_Capacity **which_entails** ‖S‖ : ℕ;

The added **which_entails** is a new construct that functions as a type of "inlined" recognition which (in this case) merely informs the type system that the cardinality of variables of type store can safely be considered as countable (specifically ℕ). Since this particular recognition was added to the **constraints** clause of a type model, this knowledge is visible to the type system wherever variables of type Store are present. In general, visibility depends on where the entails clause was added (for example, the entails clause on the module level **requires** is global to the concept as well as its enhancements and realizations). Naturally, **which_entails** assertions must be verified, so this particular example raises the following (concept level) verification condition:

Max_Capacity ∈ ℕ, ‖S‖ ≤ Max_Capacity ⊢ ‖S‖ ∈ ℕ

which follows directly from the antecedents. Further examples of such clauses will reappear throughout the remaining chapters, while the notion of countability—and associated typing ramifications—will be revisited in the context of multiset theory later in Chap. 6.

The operations shown for the most part involve straightforward manipulations of the store. For example, the Add operation **updates** some existing store S with a new key, k (assuming the key was not already present—as per the **requires** clause), while the second operation allows users to extract an arbitrary key from S (assuming the passed store is not empty).

4.3.2 Transforming Capability.

To illustrate a concrete program using the search store concept, consider the following enhancement that allows users to apply a function to each key in given store.

— RESOLVE ————————————————————————————————

```
Enhancement Transforming_Capability (
    Def T_Fn : Key ⟶ Key) for Search_Store_Template;

  Operation Transform (updates S : Store);
    ensures S = App(T_Fn, #S);
```

```
end Transforming_Capability;

Realization Iterative_Realiz (
    Operation Transform_Key (restores k : Key) : Key;
        ensures Transform_Key = T_Fn(k);
    )
  for Transforming_Capability of Search_Store_Template;

  Procedure Transform (updates S : Store);
    Var Temp_Store : Store;
    Var x : Key;
    While Key_Count(S) /= 0
      maintaining App(T_Fn, #S) = Temp_Store ∪ App(T_Fn, S);
      decreasing ‖S‖;
    do
      Remove_Any(x, S);
      x := Transform_Key(x);
      If Not Is_Present(x, Temp_Store) then
        Add(x, Temp_Store);
      end;
    end;
    S :=: Temp_Store;
  end Transform;
end Iterative_Realiz;
```

RESOLVE

The enhancement takes a generic (uninterpreted) function `T_Fn : Key` $\longrightarrow$ `Key` as a parameter and exports a single operation, `Transform`, which applies `T_Fn` over the keys of the incoming store, `#S`. The postcondition in this case is easy to express using the `App` operator discussed in Sect. 4.2.2.

Next, turning to the realization, note that it is similarly parameterized—only in this case by a programmatic operation that carries out (or, "applies") the abstract transformation function (`T_Fn`) passed into the enhancement's interface. The code itself works as one might expect: it iterates while the store `S` is nonempty, calling `Transform_Key` each iteration to transform the most recently removed key, `x`—placing it into `Temp_Store`. Once finished, we swap the temporary store with `S`, thus satisfying the **`updates`** mode associated with `S` in the specification.

4.3.3 Generating Verification Conditions for `Transform`

Upon loading this program in the VGui tool, users are presented with the screen in Fig. 4.5.

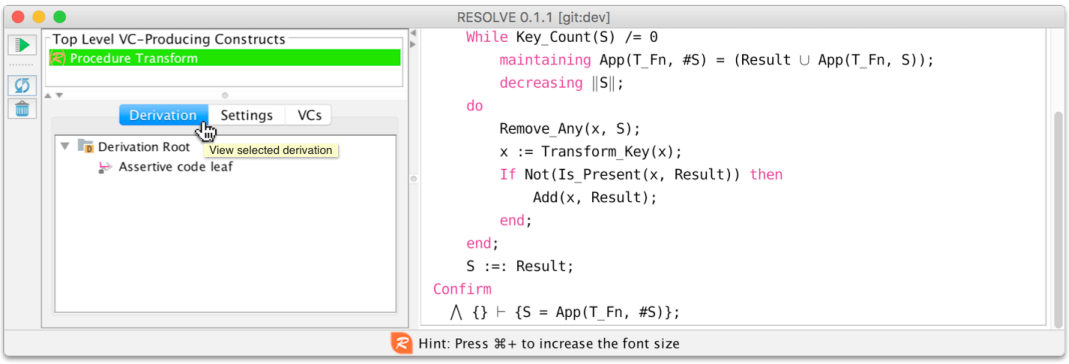


Fig. 4.5: The VGui tool loaded with an iterative realization of the `Transforming_Capability`

The workflow for using the tool mirrors those for other autoactive languages such as KeY and Why3. That is, users write code and specifications in a third party text editor (such as Vim or Atom), then, when ready to verify or generate VCs, they can load their work into the interface through either the command line (by compiling with the `-verifiergui` flag) or through the interface itself (by selecting **File** → **Load** or clicking 📂 in the tool bar).

Tool Overview: Error Reporting, Derivation Trees, and Settings.

If errors are present in the component being loaded (syntax, type, or otherwise), they are reported along with any relevant details through an error notification dialog (Fig 4.6).

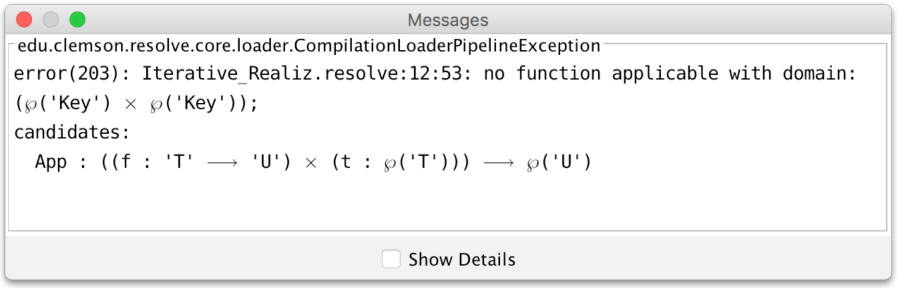


Fig. 4.6: VGui error dialog indicating a type error

Once successfully loaded, users are presented with the screen shown in Fig. 4.5. Since the

current example consists of only a single procedure, the **VC Producing Constructs** view contains only a single item (identified with the icon).

Pressing automatically generates simplified VCs from each verification producing construct selected (which, by default, is all of them). Pressing reloads the current session when/if a change is made to the loaded component. Since the reload action is used most frequently in the process of verifying code, where frequent tweaks to code or specifications are usually necessary, the reload action automatically regenerates VCs for any previously derived VC-producing constructs.

The **Derivation** tab window immediately below tracks the derivation steps of the currently selected construct as a tree, while the large pane on the right is updated with the contents of the currently selected derivation tree node: namely, a partially derived assertive code block.

The **Settings** tab allows users to tweak prover and VC-generator related settings such as, e.g., changing the prover's timeout or bounding the number of simplification rules that may be applied on any given step of the derivation (Fig. 4.7).

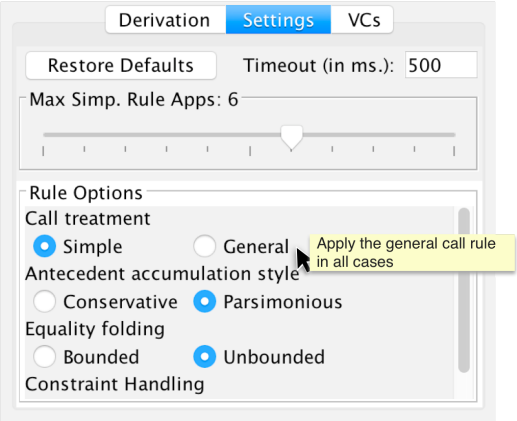


Fig. 4.7: The VGui settings tab

Other options include the ability to control how aggressive the verifier's simplifier is in folding/substituting equalities of the form $v = trm$ (where $\mathsf{IsSyntacticVar}(v)$) or how to handle the addition of antecedents to the various sequents within a final confirm assertion. We note that each rule setting is given (what we consider to be) a sensible default value that treads the line between more verbose VCs and ones with fewer extraneous givens. The average user will likely only need to tweak the prover timeout setting.

Math Type Tooltips.

Hovering the cursor over any wffs or their subterms within the sequents accumulated in the final confirm thus far (or, alternatively, in the final VCs) presents users with tooltips that show the mathematical type of the selected term (Fig. 4.8):

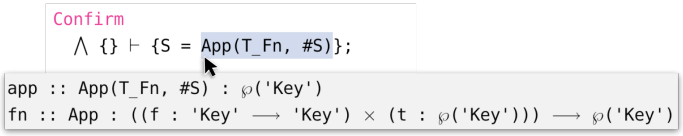


Fig. 4.8: An instantiated math type tooltip for an application of `App`

Note that in cases of function applications where the underlying function's signature contains type variables (such as `App`, Sect. 4.2.2), the tooltip shows the argument-instantiated version of the signature.

Deriving VCs.

To demonstrate the VC-generation process, we apply several rules interactively, detailing the transformations that take place. For convenience—and for reasons of brevity—we do not provide the full formalizations of the rules we're applying (consult [101] for this). Further, we also assume the statement rules for swap and while have already been applied, leaving us initially with two separate branches of assertive code:

- One in which the while statement's path condition ("PCnd") is true $\|S\| \neq 0$,
- and another in which it is false: $\neg(\|S\| \neq 0)$.

As suggested by the selected tree node (in the top left of Fig. 4.9), we consider the branch where the condition is true.

In Fig. 4.9 (top), hovering over the formula in the first and only sequent and clicking brings up a rule selection menu that allows us to apply the AndRight sequent reduction rule—which splits the sequent (shown under). Here, sequent-based rule applications are signified with a icon in the derivation tree, while program statement proof rule applications are signified with a S icon. Upon any rule application, sequent or statement, the tool automatically refocuses the window to the

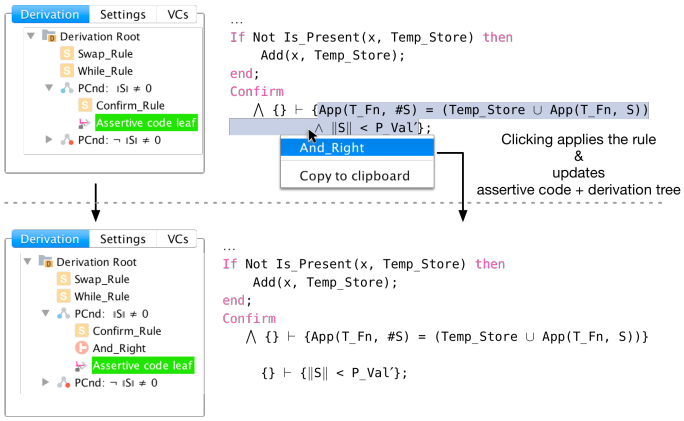


Fig. 4.9: Eliminating the ⋀ conjunct in the final confirm

most recently derived assertive code leaf. Note that a derivation leaf with any statement rules still applicable is called *open* and is signified by the icon in the derivation tree.

Next, we apply the relevant conditional rule to eliminate the if-then statement as follows:

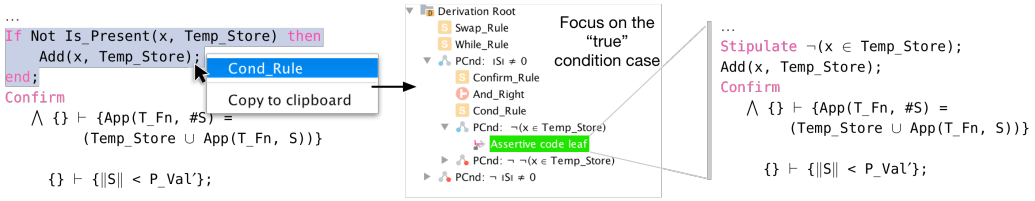


Fig. 4.10: Eliminating the if-statement; notice that the program expression for the condition `Not Is_Present(x, Temp_Store)` is converted into the appropriate "math" version: `¬(x ∈ Temp_Store)` using the functional contracts for `Not(..)` and `Is_Present(..)`

After applying the rule for if-statements, the call to `Add(x, Temp_Store)`—originally found within the body—is pulled out and placed immediately before the final confirm. Next, when one clicks the call to `Add`, the rule option menu provides two possible rule variants that can be applied (see Fig. 4.11, left).

The first, termed the "simple call rule," can be applied to any call whose postcondition is specified in an explicit style. To have an explicit style postcondition, each parameter with a mode of **`updates`** or **`replaces`** must be expressed strictly in terms of their `#`-denoted incoming values and potentially the incoming variables values of other parameters as well. Thus, the postcondition for `Add`:

```
//where k has mode restores and S has mode updates
ensures S = #S ∪ {k};
```

is indeed explicit, as its sole conjunct is expressed as an equality relating the outgoing `S` to the the union of the incoming `S` and the singleton `{k}`.

The second "general" version of the rule can be applied to calls with either an explicit or implicit style postcondition. An example of an implicit style specification can be observed in the postcondition of `Remove_Any`,

```
ensures k ∈ #S ∧ S = #S ∼ {k};    //where k has mode replaces
```

The specification is implicit since the first conjunct describes the outgoing value for `k` in relational terms of what it could be (i.e., any value in `#S`), as opposed to mapping it to any one explicit output value.

Implicit vs. Explicit. The simple version of the call rule is usually preferred whenever applicable since blindly applying the general version tends to introduce additional intermediate (primed) variables which—in the context of larger procedures—can add significant clutter to resultant VCs. Beyond appearances, however, [45] finds no immediately obvious correlation on the provability of code involving specifications written in a strictly implicit vs. explicit style. Nevertheless, this question likely requires further study with larger and more intricate components in order to draw stronger conclusions.

For the purposes of the current example, we apply the general version of the call rule as it highlights some additional intricacies involving the accumulation of antecedents.

Applying the rule results in the following assertive code (Fig. 4.11):

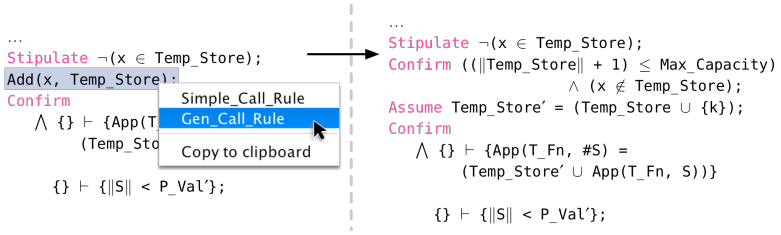


Fig. 4.11: Applying the general call rule

As per design-by-contract, to verify the call's correctness we: (i) **`Confirm`** that the argument-specialized precondition of `Add` is true at the site of the call, and (ii) **`Assume`** that the

postcondition holds afterwards (which intuitively makes sense—as we're the callee, not the implementer). These verification statements have the following behavior when processed:

- First, the **`Assume`** $\varphi_1 \wedge \ldots \wedge \varphi_n$ statement adds each φ_i as an antecedent to any sequent in the final confirm whose existing set of assertive free variables overlaps with $\mathsf{AFV}(\varphi_i)$. For example, the **`Assume`** equality shown in Fig. 4.11 (right) will be added as an antecedent to the first sequent, but not the second.

- Second, the **`Confirm`** $\varphi_1 \wedge \ldots \wedge \varphi_n$ statement, once processed, merely adds a new sequent of the form $\vdash \varphi_1 \wedge \ldots \wedge \varphi_n$ to the final confirm. This, in some sense, represents the creation of what will become a VC.

Any logical connectives introduced into the final confirm assertion's sequents after processing the assume and confirm statements are eliminated using the sequent reduction rules shown in Fig. 4.2.

The last statement we discuss in this example is **`Stipulate`** [101], which recall was introduced after processing the if-statement. This functions as a (stricter) variant of **`Assume`**, in that it unconditionally adds the stipulated formula to the antecedent of each sequent in the final confirm (regardless of whether or not there are overlapping assertive free variables).

This is done to preserve contradictions that may arise, e.g., when processing nested conditionals whose outer path conditional precludes any inner conditional. By stipulating such path conditions, it ensures they remain present among the antecedents of a particular VC. This, in turn, gives the verifier the chance to detect contradictions up-front during proof search—resulting in faster, more more precise feedback for users.

Completing the Derivation and Examining Steps.

At this point, we complete the remainder of the derivation automatically by pressing the derive (▶) button. Once complete, the remaining statements in each branch are eliminated automatically and closed (signified by the "closed" icon in the GUI).

Viewing the Rules and their concrete instantiations. Users can observe any of the steps taken in detail by clicking an internal node of the derivation tree (Fig. 4.12):

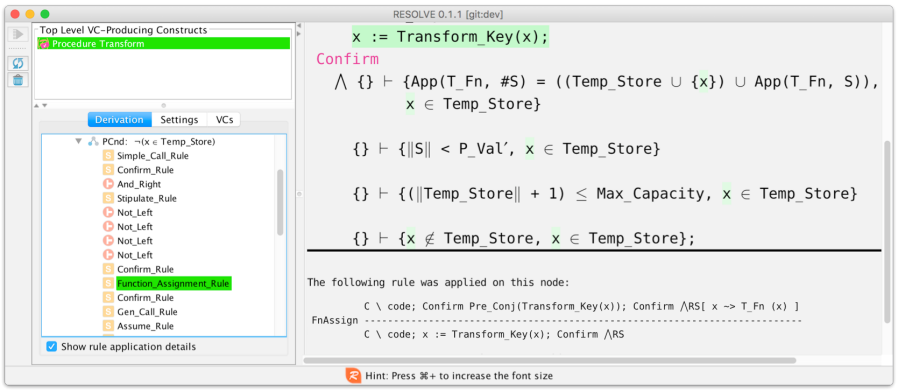


Fig. 4.12: Viewing rule application details on an inner node of the derivation tree

The site of a rule's application is highlighted in dark green, while any subterms throughout the final confirm assertion that are affected by the rule are highlighted in a lighter shade of green. Further, the "Show rule application details" checkbox gives more advanced users the option of seeing the underlying rule's formal schema—augmented with certain details specific to the context (e.g., in Fig. 4.12 the formal rule for processing function assignment statements is shown instantiated with the actual concrete terms being substituted).

4.3.4 Verifying `Transform`

Once the derivation is complete, switching to the **VCs** tab, we can peruse the resulting VCs and attempt to verify them by pressing the "prove" button (labeled with a ▶ icon—not shown). After being pressed, the same button can be pressed again to cancel the attempt.

Any successfully proven VCs are displayed with a ✅ icon, while any that timeout are given a ⚠ icon. If there is an internal error, or the verification attempt is cancelled, any remaining VCs are given a ⛔ or a ⚠ icon, respectively.

To illustrate some of steps in reasoning about set-based VCs, we examine VC #1 (shown in Fig. 4.13) which corresponds to proving that the code satisfies the operation's postcondition.

First, using given (1) in conjunction with the following empty-set corollary:

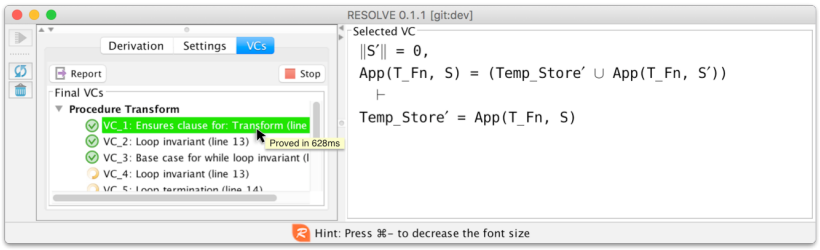


Fig. 4.13: Invoking the prover on VCs generated from an iterative realization of the transforming enhancement

$$\forall\ C\ :\ Cls,\ (\|C\| = 0) \iff (C = \emptyset)$$

we can simplify given (2) of the original sequent as follows:

$$\|S'\| = 0,$$
$$App(T\_Fn,\ S) = Temp\_Store' \cup App(T\_Fn,\ \emptyset)$$
$$\vdash$$
$$Temp\_Store' = App(T\_Fn,\ S)$$

Then, using `App_C1` (from Sect. 4.2.2), we can further simplify given (2) to obtain:

$$\|S'\| = 0,$$
$$App(T\_Fn,\ S) = Temp\_Store' \cup \emptyset$$
$$\vdash$$
$$Temp\_Store' = App(T\_Fn,\ S)$$

At this point, applying the lemma: $\forall\ C\ :\ Cls,\ C \cup \emptyset = C$, followed by a single substitution of $Temp\_Store'$ into the succedent completes the proof.

Given suitable theory developments, many of the remaining VCs are similarly straightforward. Consider, for example, the following VC which corresponds to verifying the base case of the while loop's invariant:

$$\vdash \quad App(T\_Fn,\ S) = (\emptyset \cup App(T\_Fn,\ S)).$$

In the next chapter we discuss the integration of the VGui tool demonstrated in this chapter into a new F-IDE named RESOLVE Studio which has been developed as part of this work.

Chapter 5

RESOLVE Studio: A Formalization IDE for Engineering Software Components

In this chapter we introduce a new Formalization Integrated Development Environment (F-IDE) named RESOLVE Studio. Starting with a discussion of the environment's architecture and some core design decisions, the chapter transitions into a demonstration of RESOLVE Studio—walking through the development of a revised version of the queue component presented in Chap. 3. In doing so, we present some of the features supported by the F-IDE, which have been designed to benefit both novice and expert users alike.

5.1 The Case for Integrated Component Development Support

As auto-active [70] approaches to program specification and proving gain traction from systems such as Why3 [38], Dafny [68], RESOLVE [100], and AutoProof [104] developers are increasingly reliant on front-end tools to help engineer formally specified components. Whether users aim to engineer new theory developments, abstract data types, or are simply writing code to meet some specification, each of these tasks when tackled individually (and especially when combined) can slow development and significantly increase cognitive burden on users at all levels of experience.

Irrespective of the approach to verification (i.e., interactive, auto-active, or fully automatic—see Chap. 2) many efforts currently (including our own) employ a mixture of compiler backed 'viewer-based' GUIs that specialize in the interpretation of verifier feedback in conjunction with third-party text editors where the changes to artifacts are made. While it has been observed (mainly through classroom usage) that this style of interaction is well suited to the interpretation of verification results [55, 105, 23] in a web-based context, it nevertheless carries some downsides when employed in a non-integrated, desktop-based context. Some issues that arise are workflow related (such as having to constantly switch between one or more tool windows and a text editor after each minor change), while others are usability related and involve the interpretation of compiler and verifier feedback (which is often far removed from the original source text, making it difficult for newcomers to pinpoint the exact source of errors in their code, specifications, or both).

Modern IDE's blunt these problems by integrating all critical aspects of the tool into a single, unified environment that goes beyond the current capabilities of RESOLVE's web-based system. Indeed, such environments combine integrated project and library management with rich editing and analysis capabilities that provide everything from responsive completions for references and keywords as users type, to powerful navigational features (such as goto declaration or find usages). Such functionality is considered nearly essential to understanding large-to-moderately sized codebases in traditional software development contexts. In this chapter we demonstrate how to extend these useful features to an F-IDE: that is, an IDE for the formalization of mathematical developments as well as for the development of formally specified components. Fig. 5.1 shows a preview of our environment, RESOLVE Studio, in action.

Development of such an environment is nontrivial, and typically requires usage of a 'heavy duty' framework such as XText[1] (for Eclipse), Microsoft's Visual Studio, or the JetBrains platform. However, even with the help of these frameworks, such an effort still carries a steep up front development cost, followed the need for long term maintenance of the code that inevitably results. Nevertheless, those in both the program verification and proof assistant communities [72, 105, 104, 19, 35] have become increasingly aware of the benefits such an investment—if designed to exploit reuse—

[1] https://www.eclipse.org/Xtext/

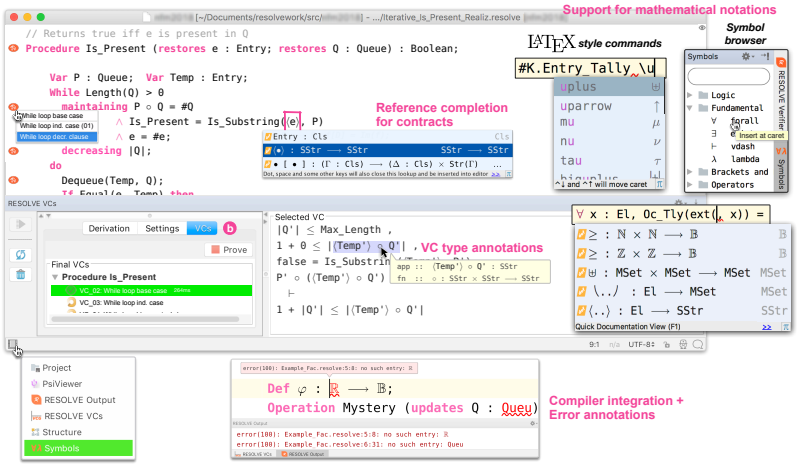


Fig. 5.1: RESOLVE Studio and some of its features

can add to long term research, scalability, user workflow, and other educational objectives. In the following section we detail the design of our own environment that we feel retains the powerful feature-set users have come to expect from modern IDEs, while remaining maintainable to developers through a design insistent on reuse whenever possible.

5.2 RESOLVE Studio Design and Architecture

The F-IDE we present in this chapter is built on the JetBrains Community Edition (JCE) source code, publicly available on Github at: https://github.com/JetBrains/intellij-community.

5.2.1 Reuse Centric Compiler Integration

From the outset of RESOLVE Studio's development, we have sought to reuse as much functionality from the current compiler as possible in the implementation of the environment's various features.

In particular, we have determined that the compiler should be responsible for the usual analysis, code generation, and typechecking tasks, while the F-IDE's primary job is to layer on additional analysis in the form of dynamic *inspections/completions*. This separation makes it pos-

sible to perform certain forms of analysis dynamically as users type: e.g., suggesting keywords and other symbols available in scope. This design is intended to keep the the F-IDE responsive to users, while at the same time avoiding unnecessary duplication of intricate compiler-specific tasks within the F-IDE's codebase (e.g., mathematical typechecking). The downside of course is that users are occasionally required to manually 're-analyze' the file being edited to ensure that specifications are type conformal—though overall this is a relatively minor concession that helps keep our research based system adaptable to change in the long term.

In cases where compilation errors (typechecking or otherwise) are present, we handle their reporting cleanly through a compiler-specific API that permits, for instance, the collection of error information (suitable for annotation in the F-IDE's main editor and output console) or the registration of observers for monitoring proof results and/or viewing information about a particular proof obligation. The two systems and their various submodules are depicted in the architectural diagram in Fig. 5.2.

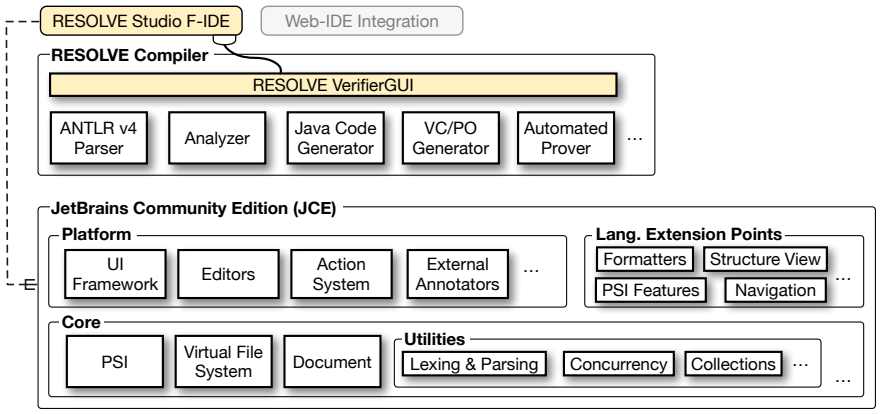


Fig. 5.2: Involved systems (from top to bottom) including (i) RESOLVE's front-end development environments, (ii) the compiler itself, and (iii) the JCE platform and its various internal components and extension points; The forked arrow connecting RESOLVE Studio to the JCE represents a version control system (VCS) dependency

Critically, since both the JCE and RESOLVE compiler run on the Java Virtual Machine—and both use the same underlying swing-based framework for UI, it was a relatively straightforward task to integrate our existing VGui tool directly into RESOLVE Studio. This connection is communicated in Fig. 5.2 by the plug shape connecting the boxes highlighted in yellow (the results of this

integration can also be seen at work in Fig. 5.1, bottom). Such reuse is not only beneficial from a maintainability perspective, but also allows us to present the same verification UI to two separate groups of users: those who prefer to work using the standalone VGui tool, and those who wish to use RESOLVE Studio. Integration of similar functionality into our web-based environment remains an area of future work.

5.2.2 A Standalone Environment

While early on we envisioned the environment as a plugin compatible with existing IDE's in the JetBrains product line (e.g.: PyCharm, IntelliJ, CLion, etc.), we instead opted to create a standalone environment for several reasons.

- **Minimalist UI.** IDEs historically have suffered from bloated and slow interfaces. By opting to create a standalone tool, we were given additional control over the structure of the IDE's default UI. Thus, to make it more suitable for educational and research usage, we reduced the number of elements on screen and within menus at any one time—prioritizing certain core tasks (such as proving) and deactivating numerous others far removed from the purposes of verification (though some can be re-enabled based on user preferences).

- **Consistency.** By disconnecting our environment from those in the existing JetBrains product line—and the rapid pace at which these IDEs are changing—we are able to provide a more consistent and ultimately more stable user experience. Indeed, the alternative involves providing research language support via a plugin across a spectrum of different IDEs—each with subtly different versions and the potential (unforeseen) compatibility issues.

- **Branding.** Though an admittedly minor point, we wanted more control over the branding and artistic aspects of the IDE. Indeed, as this is a long term research project, used regularly by researchers and students for component development purposes, it made sense to invest in some distinctive artwork (such the logo or F-IDE's splash screen shown in Fig. 5.6).

To achieve these aims, we forked a recent stable release of the JCE from GitHub, within which we created a new, self contained package: `com.jetbrains.resolvestudio` for housing

the code that drives RESOLVE Studio. And since this package depends only on features of the JCE that are mature and well established—we are able to regularly merge updates from the company's official "upstream" repository with minimal merging conflicts.

5.2.3 Rich Editing, Document Markup, and Dynamic Inspections

The JCE platform represents program text via *Program Structure Interface* (PSI) trees that are capable of providing a complete syntactic and semantic view of a codebase. PSI trees are kept up to date as users type, and even support sophisticated forms of analysis on partially constructed (syntactically erroneous) trees. Thus, to provide language support that includes the advanced features users expect (such as completions), it is strictly necessary to have a PSI built for targeted language. There are a number of ways to do so, some of which we discuss in this section.

Handling Parsing: Building a PSI. Since the newest version of the compiler uses ANTLRv4 for parsing, it was necessary to connect this to the JCE's PSI-based parsing infrastructure. Unfortunately, the JCE does not provide native facilities for constructing PSI trees from ANTLR. Rather, the JCE uses a parsing library called GrammarKit[2] wherein users provide a `*.bnf` grammar combined with a JFlex lexer. From this, GrammarKit automatically generates a PSI tree for the targeted language.

Fortunately, there are some notable similarities between ANTLR and GrammarKit: e.g., they both accept EBNF-notated grammars as input, and (more importantly) both support left recursive rules which greatly simplify the translation of RESOLVE's more involved syntactic constructs (such as expressions/terms—the rule for which we summarized in the previous chapter).

Given the similarities, it was relatively easy to create a simple syntax-directed translation tool to convert one grammar format to the other (this process is illustrated at a high level in Fig. 5.3). The benefits of such a translation are notable, as it helps keep the grammars for the compiler and F-IDE in sync—while at the same time yielding a PSI structure for RESOLVE with all of the capabilities this affords.

[2] https://github.com/JetBrains/Grammar-Kit

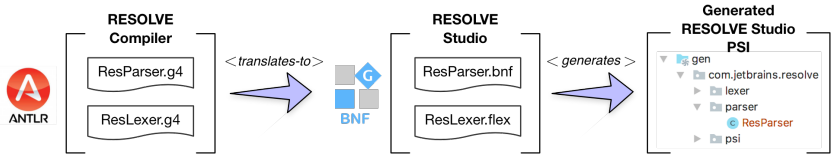


Fig. 5.3: Steps for converting RESOLVE's parsing infrastructure

Why not XText? Unfortunately, it was not possible to use XText given that it doesn't support the newest version of ANTLR (ver. 4) and all the advantages it holds over prior versions (see the introductory chapters of [88] for a summary of some). It's worth mentioning as well that there is a library (started by the creator of ANTLR—Terrence Parr) that attempts to convert ANTLRv4 syntax trees to PSIs, though it is not well maintained, and (in our experience) fails to produce a PSI structure of the same quality as one produced natively by GrammarKit.

Live Templates and Keyword Completions. With the support of a dedicated PSI structure, we immediately gain access to a number of editing features such as context sensitive completions for "live-templates" as well as keywords—both of which are triggered only when the underlying PSI is in a (developer) specified state. An example of a live template is shown below in Fig. 5.4.

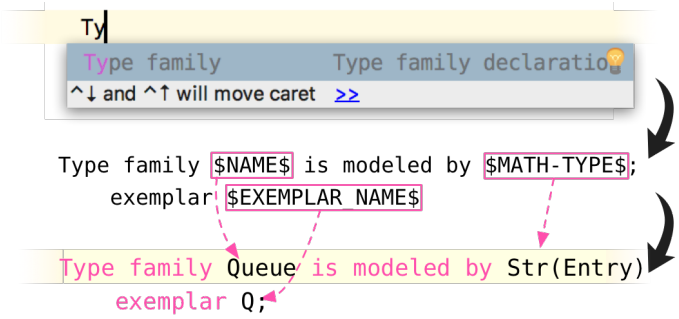


Fig. 5.4: A live template for a model type declaration; 'holes' in the template are enclosed in pink boxes

Live templates are essentially "documents with holes" that come in handy for the more wordy language constructs that are prone to being mistyped. For example, the model type template shown in Fig. 5.4 is only suggested by the F-IDE (i) if the user is editing in the global context of an interface module (i.e. a concept or enhancement) and (ii) if the user types `T` (alternatively, one can also press `Alt` + `Space` to query for any applicable keywords, references, or live templates without typing anything). Once the template is inserted, users can systematically fill each hole—

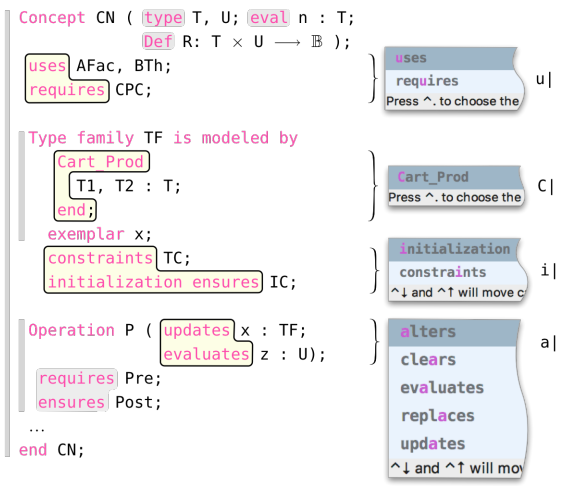


Fig. 5.5: A concept schema showing both complete-able keywords as well as available live templates

pressing the Tab key to automatically reposition the cursor to the next unfilled hole.

More than just larger constructs, the F-IDE also supports contextual keyword completions. The completions are contextual since they are not suggested in cases that would result in an erroneous PSI. For example: under operation `P`, the F-IDE will not prompt the insertion of a second `requires` after the first is already present (similarly it also avoids the suggestion of putting `ensures` before `requires`—as this is not syntactically valid input). Fig. 5.5 shows various available completions, with the initiating sequence of characters appearing on the far right.

Here, the availability of a live template for a particular construct is indicated by the vertical grey bars on the left next to the larger declarations including: the module itself `CN`, the type family `TF`, and the operation `P`. Keywords supporting completion are are highlighted in yellow colored boxes and can be triggered by typing the character shown on the right. Keywords appearing within lighter grey (non-outlined) shapes can also be completed, though we elide them for reasons of space.

5.3 Developing a Queue Concept in RESOLVE Studio

After launching RESOLVE Studio, users are presented with the following screen (Fig. 5.6). Clicking the "Create new Project" button ✸ opens a panel (Fig. 5.7) wherein users provide both a name for their project (in this case, we simply call it `queue`) and a location pointing to the directory containing the RESOLVE compiler (in this case, version `0.1.1`).

Fig. 5.6: The F-IDE's splash and landing screens

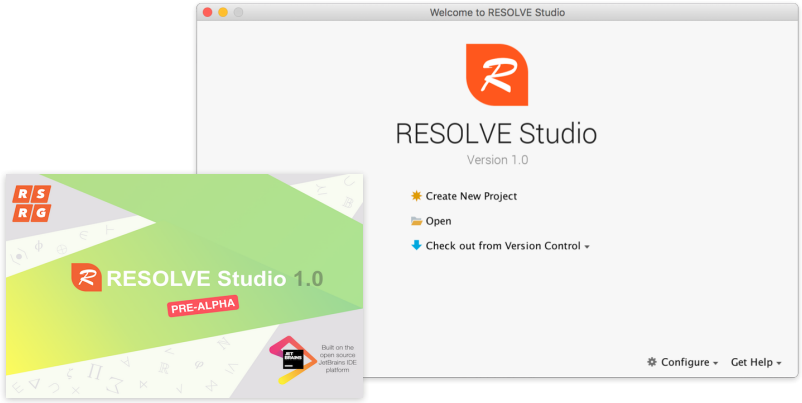
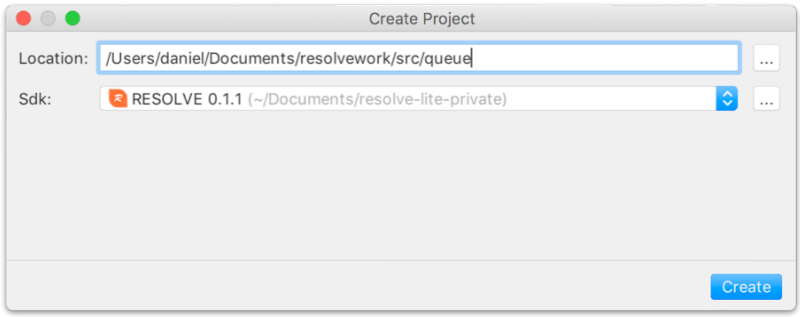


Fig. 5.7: Setting up a new project



5.3.1 Project Structure and Organization

When opened for the first time, the F-IDE starts in minimalist (folded) form. Hovering the cursor over the ▣ icon in the environment's lower left hand corner (Fig. 5.8, left), then selecting **Project** attaches the usual project explorer to the left hand side of the environment's window. Alternatively, clicking the ▣ icon arranges the shown tools as buttons along the panes of the window (though in this work we typically keep them deactivated to reduce visual clutter).

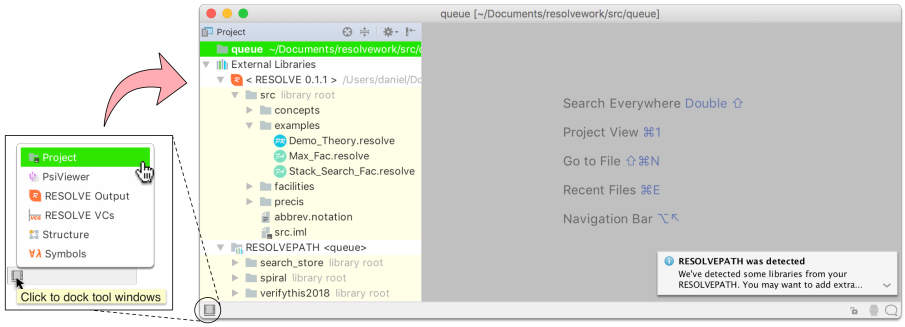


Fig. 5.8: Opening the environment and performing first-time setup of the project window

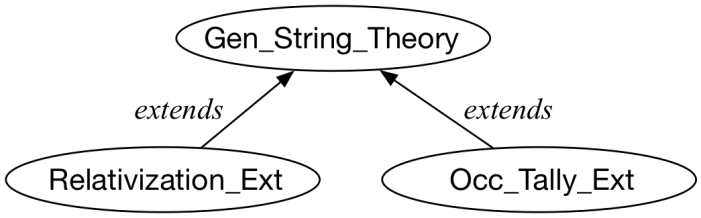
The environment itself adheres to a fairly traditional layout: i.e., project files are placed under the queue folder (which is currently empty), while the bottom half (under "external libraries") provides access to the RESOLVE compiler's core library of reusable components, as well as third-party projects on RESOLVEPATH—a distinguished directory where all user projects are placed. The environment keeps track of the files under these directories, and the project pane is updated in realtime to reflect any changes to their contents, such as additions or removals, etc.

Mathematical Support: General String Theory.

Before we demonstrate usage of the F-IDE for engineering concepts like `Queue_Template` , we first consider a generalization of string theory which we have formulated in RESOLVE Studio and have added to RESOLVE's standard library as part of this work.

Informally, our general theory of strings consists of a mathematical class `SStr` of all possible strings (including those with heterogeneously typed elements), the empty string Λ, and an extension function `ext` that adds an element to the end of a string.[3] The organization of the updated theory and some of its extensions are shown in Fig. 5.9. We defer discussion of `Occ_Tally_Ext` until the next chapter, where we use it in the realization of a generic sorting component.

Fig. 5.9: String theory and its extensions



The top level precis introduces string operators defined over `SStr`, including concatenation ($\circ$: `SStr` $\times$ `SStr` $\longrightarrow$ `SStr`), length ($|\bullet|$: `SStr` $\longrightarrow \mathbb{N}$), and others (See Fig. 5.10 for their definitions).

Note that inductively defined functions consist of two parts: a base case `(i)` and an inductive step `(ii)`.

The `Relativization_Ext` precis (Fig 5.11) introduces a string formation operator param-

[3] Not to be confused with `cons`, which adds to the front of an ordered collection and is common in functional languages

Fig. 5.10: A snippet of `Gen_String_Theory` (taken from RESOLVE Studio)

Gen_String_Theory.resolve

```
Inductive Def (α : SStr) ∘ (β : SStr) : SStr is
    (i.) α ∘ Λ = α;
    (ii.) ∀ v : El, α ∘ ext(β, v) = ext(α ∘ β, v);
Corollary C1: Is_Associative(∘);

Inductive Def |(α : SStr)| : ℕ is
    (i.) |Λ| = 0;
    (ii.) ∀ x : El, |ext(α, x)| = suc(|α|);

Corollary L1: ∀ α : SStr, |α| = 0 ⟺ (α = Λ);

Corollary L2: ∀ α, β : SStr, |α ∘ β| = |α| + |β|;

Corollary L3: ∀ α, β, γ, δ : SStr,
    (α ∘ β = γ ∘ δ ∧ |β| = |δ|) ⟹ (β = δ ∧ α = γ);
```

eterized by a generic class, `Str`$(\Gamma : \texttt{Cls})$, which restricts the type of its entries to Γ.

Fig. 5.11: A relativization extension for strings (taken from RESOLVE Studio)

Relativization_Ext.resolve

```
Precis Relativization_Ext extends Gen_String_Theory;

    Def Str(Γ : Cls) : 𝒫(SStr) ≜
                    {α : SStr | Occ_Set(α) ⊆ Γ};
        Recognition ∀ Γ : Cls, Λ : Str(Γ);
        Recognition ∀ Γ : Cls, ∀ s : Str(Γ), s : SStr;
        Corollary PS1: Str(El) = SStr;
            ...
```

This extension also registers a number of **`Recognition`**s that permit, for example, users to pass a term $s : \texttt{Str}(\mathbb{N})$ where a term of type `SStr` is expected (see the second recognition in. Fig. 5.11).

We defer discussion of RESOLVE Studio's mathematical editing features to the next section, where we detail the construction of the queue concept interface that utilizes both the general theory and its relativization extension.

5.3.2 F-IDE Assisted Development of `Queue_Template`

Consider the screenshot in Fig. 5.12 of a (partially completed) `Queue_Template` that we've added to the project.

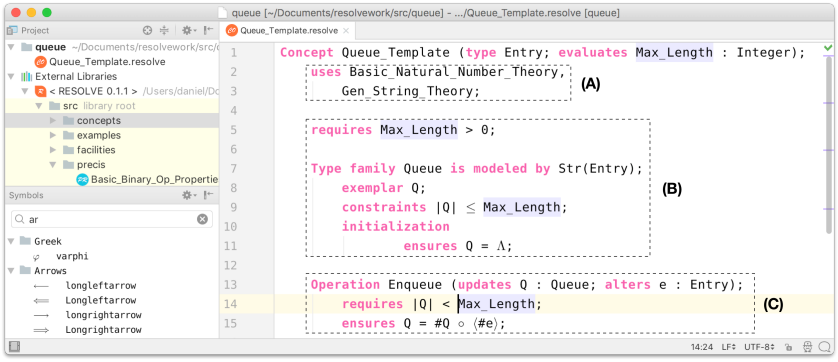
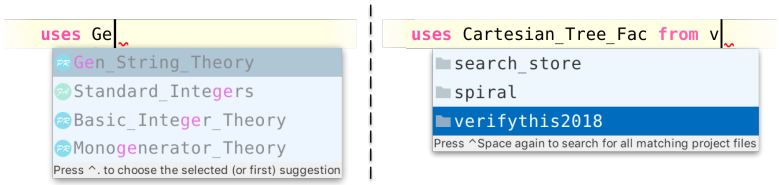


Fig. 5.12: Constructing a portion of `Queue_Template` in RESOLVE Studio

Our discussion follows the labeled sections (some of which contain errors) outlined in Fig. 5.12. We emphasize that the goal here is not to provide a comprehensive summary of the features provided by RESOLVE Studio, rather, to give readers a general impression of the editing experience the environment offers.

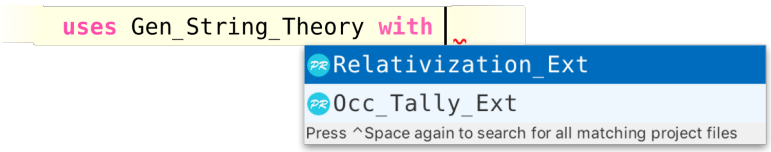
(A) Assistance with Module Imports. When typing `uses` lists, RESOLVE Studio automatically suggests available modules (see Fig. 5.13, left).

Fig. 5.13: Uses completions for modules (left) and user-defined projects (right)



Syntactically, module references within a `uses` list are represented as a single identifier. The F-IDE will automatically search both the current project as well as the compiler's core collection of modules—suggesting any with names that are exact or partial matches. To import user-defined components on RESOLVEPATH, users can specify a `from` clause identifying the project containing the desired module(s). The F-IDE assists with this (Fig. 5.13, right) and will also suggest any applicable theory extension modules following a `with` clause (Fig. 5.14).

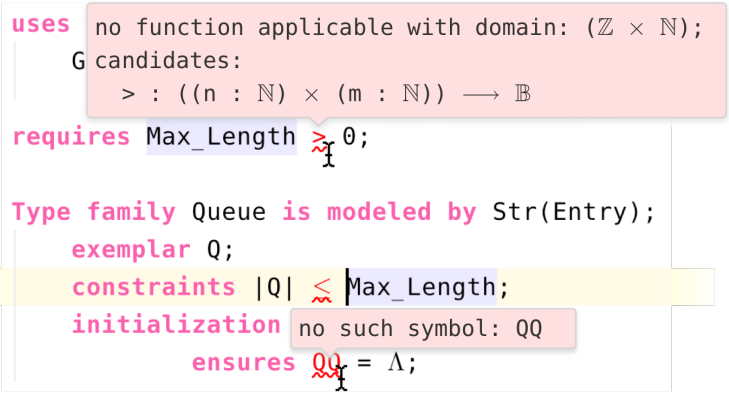
Fig. 5.14: Smart completions for theory extensions



In this case, we need the generic operators introduced in the string relativization extension from Fig. 5.11 to specify our similarly generic queue.

(B) Type Checking and Analysis via Compiler Feedback. After typing the text shown in (B), attempting to analyze the current file (by pressing Cmd + Shift + A or by right clicking in the editor and selecting **Analyze**) results in several errors, which are annotated directly into the editor (Fig. 5.15).[4]

Fig. 5.15: Examining type error annotations



Hovering the cursor over the errors we see that the first two involve types, while the third arises from a from a simple typo. Hovering over the topmost error, we learn that the first argument to > (namely, `Max_Length : Integer`) has a math type of $\mathbb{Z}$ (as this is the model for type `Integer`). And since integer theory is not imported—and relational operators from natural number theory all operate over $\mathbb{N}$—the system cannot find a suitable operator for > (this is also the problem with $\leq$ in the `constraints` clause).

However, since the module level precondition asserts that `Max_Length` must be strictly positive, we can augment the specification as follows:

```
requires Max_Length > 0 which_entails Max_Length : ℕ;
```

[4] All error messages are also echoed into an output window docked at the bottom of the IDE (not shown here)

As discussed in the previous chapter, the 'entails' clause is used to inform the type system that the variable term `Max_Length` can also be considered to be of type $\mathbb{N}$ within the scope of the concept and its realizations. This specific refinement of course raises the following (trivial) concept-level proof obligation:

$$\texttt{Max\_Length} > 0 \quad \vdash \quad \texttt{Max\_Length} \in \mathbb{N}.$$

For the purposes of minimizing the search space in automated proof finding, it's generally desirable to limit the number of domains being used in the specifications of a given concept—as these specifications (and any verification-stymieing intricacies involved in their typings) ultimately end up in proof obligations generated from client code.

The last remaining error is easy to fix (simply change `QQ` to `Q`), though we use it to elaborate on how the F-IDE handles error annotations in a constantly shifting sourcefile (see Fig. 5.16 below):

Fig. 5.16: Persistent editor markups for analysis results

Specifically, warning and error annotations produced by the compiler don't simply disappear once a file is edited, rather, the lines affected by a particular edit are underlined in a yellow squiggly line denoting a potential fix. Such annotations stay attached underneath the relevant source text—even if moved around (unless of course the line is erased entirely, in which case the annotation disappears). Once fixed, the yellow annotations are removed on the next analysis attempt.

(C) Editing Support. In addition to compiler-derived editor annotations, RESOLVE Studio also supports contextual reference completions that occur while users are typing. This is especially helpful when typing specifications, see Fig. 5.17.

Here, we see a sequence of completions for filling in the `ensures` clause of `Enqueue`. First, mathematical symbols can be inserted quickly by typing a backslash, which brings up a completion

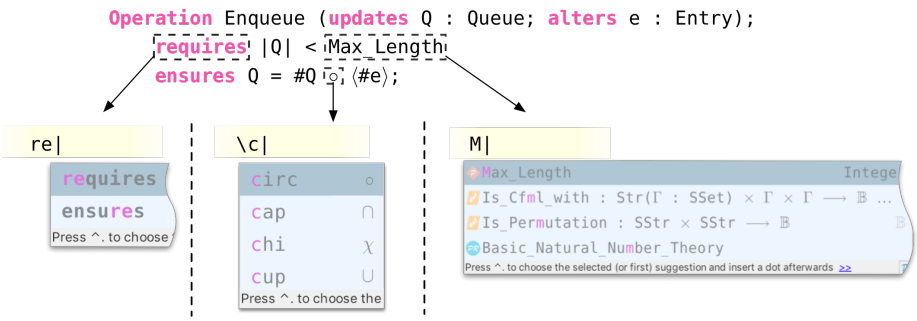


Fig. 5.17: Completions for keywords (left), symbols (middle), and references (right)

menu that is filtered as users type the desired LaTeX command.[5] Once selected, the symbol can be inserted at the cursor's position by pressing `Tab` (or `Return`). Such symbols can be inserted anywhere in the document (including within comments). Alternatively, users can 'ping' the current scope for available symbols or operators without typing anything by pressing `Ctrl` + `Space` .

The completion suggestions are designed to respect both the current scope and the semantic context in which the users are typing, as well as the divisions RESOLVE makes between mathematical and programmatic entities. For example, in operation specifications the system will suggest certain programmatic entities (such as formal parameters), while others are expressly disallowed (such as operation calls). This can help avoid common pitfalls such as, for example, a new user who might be tempted write a postcondition asserting `Length(Q) = 0` when in fact it should be expressed as `|Q| = 0`.

To facilitate larger developments, our current implementation of reference completion also allows users to navigate to declarations across files and even projects by hovering the cursor over a symbol, holding down the `Ctrl` key, then clicking (this can be accomplished through a menu item as well).

Dynamic Intention Actions. Beyond completions, RESOLVE Studio also supports a more sophisticated (dynamic) form of analysis called *intention actions* that are discovered via background analysis performed on the PSI while users are typing. Intentions actions can be used to point out

[5] If one is not familiar with LaTeX and its commands, a symbol browser can be docked along the F-IDE's window; see Fig. 5.1, upper right

potential and actual flaws in both code and specifications—then prompt users for appropriate fixes that the F-IDE will then automatically perform. For example, one intention for specifications analyzes `While` loops, and suggests a `changing` list based on the variables present in the body of the loop that are being modified, specifically:

- variables that appear on the left hand side of an assignment statement,
- variables that appear on either side of a swap statement,
- and variables that are passed to operations with a mode of `alters`, `clears`, `replaces`, or `updates`.

Once detected, the affected region of code is highlighted in either yellow or red based based on the found issue's severity. In this case, since a malformed or incomplete `changing` clause can affect soundness, a red bulb  is placed next to it indicating the availability of a *quick-fix* that, once clicked, inserts the relevant variables into the `changing` clause.

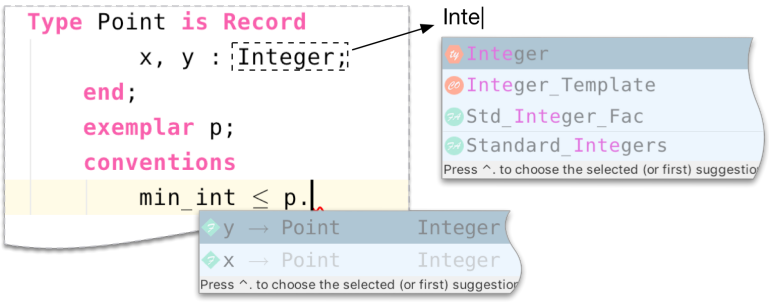
JetBrains based IDEs for existing popular languages such as Google's Go come pre-packaged with many intention actions (for example, GoLand,[6]—the JetBrains IDE targeting Go—currently includes ∼70 intention actions, while IntelliJ for Java has well over 150). An immediate direction for future work involves the creation of additional intentions aimed at addressing common flaws in specifications (such as misuse of parameter modes). These can be added to RESOLVE Studio incrementally by simply extending a predefined interface in the JCE and thus would make for an ideal (and relatively self-contained) summer research project for the motivated undergraduate or graduate researcher.

[6] https://www.jetbrains.com/go/

5.4 Writing and Executing Queue Client Code

The F-IDE's features also apply to ordinary code as well. To illustrate, we create a new client facility module and a define a local record type, `Point` (Fig. 5.18).

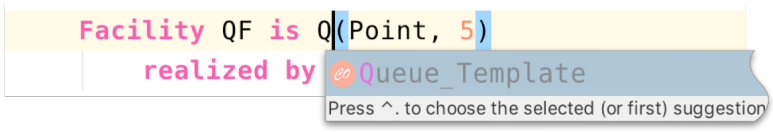
Fig. 5.18: Defining a program record type



The figure shows off completions for programmatic types (such as `Integer`) and record types. Recall that `conventions` clauses must hold before and after each externally visible procedure body, and that even seemingly programmatic expressions like `p.y` can be referenced in such an assertion since each field carries a mathematical counterpart: in this case, $\mathbb{Z}$. The mathematical type of `Point` is to be interpreted then as $(\mathbb{Z} \times \mathbb{Z})$. Note too that completions will occasionally recommend a module, such as `Integer_Template`. When such a module is chosen, its name is automatically inserted into the document followed by a `::` qualifier—which in turn, triggers another round of completions for any relevant symbols appearing within the specified module.

Next, we define a simple facility `QF` that instantiates the queue concept with the local `Point` data type (Fig.5.19 below):

Fig. 5.19: Defining a facility for queues



Facility declarations benefit greatly from reference completion. Namely, when filling the slots for modules, the completions triggered are designed to suggest only concept modules after the `is` keyword and only realizations for the chosen concept after the keyword phrasing: `realized by`. This functionality works similarly for any enhancements that might be layered on `QF` as well.

Lastly, we construct a main function and write some queue client code that operates on

points (shown in Fig. 5.20).

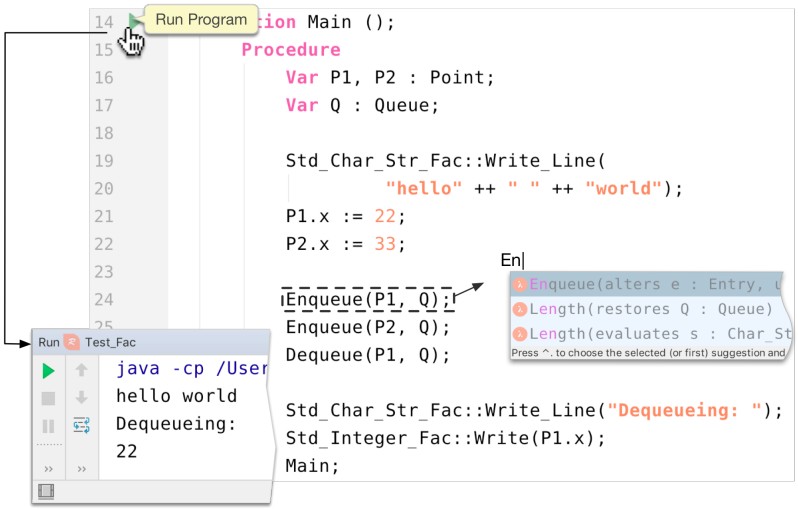


Fig. 5.20: Executing facility client code

Users can execute this code by creating a run configuration for the current facility. Such configurations persist across IDE sessions—pairing the name of the facility to be executed along with any optional user-supplied arguments. Alternatively, once a main operation is present, RESOLVE Studio will automatically detect it and place a ▶ button in the gutter. Pressing this generates the required run configuration on the fly, places the translated `.java` output files to a special `/out/` directory, and executes the resulting code using the Java compiler. This allows users to quickly execute their code with a single click.

The next chapter examines verification-related features offered by RESOLVE Studio with a particular focus on the integration of the VerifierGUI tool.

Chapter 6

Application and Evaluation

In this chapter we use RESOLVE Studio to construct a component-based system for prioritizing generic entries and employ it in a simple application. Each artifact involved is designed to be reusable, with object oriented interfaces built to encapsulate non-trivial data structures and algorithms. The chapter opens with an overview of the system, then demonstrates usage of the F-IDE through the specification and implementation of a fully generic sorting enhancement for queues.

The second half of the chapter involves the development of a new theory of multisets which we add to RESOLVE's existing collection of math units, along with the specification and implementation of a generic prioritizing concept. The prioritizer's concept and the specific realization we showcase bring together both the generic sorting enhancement discussed in the first half of the chapter, as well as a new (user-defined) theory of multisets which we use to model the prioritizing component and its various operations.

The chapter concludes with a small OS task scheduling case study (similar to some of the work presented in [30]). We also discuss some preliminary observations on the usage of RESOLVE Studio in the context of Clemson's graduate level programming languages course, CPSC 8280, where students used it to construct a small mathematical theory.

6.1 Overview: Reusable Software Development

To illustrate the notions of engineering reusable concepts and employing them in component-based implementations, we consider the design of a general sorting concept called the `Prioritizer`. In its design, we employ the algorithms as objects "recasting" approach detailed in [110] to yield a concept that is:

- Parameterized by a generic, binary predicate that captures the desired ordering relation.
- Modeled as a two-phase machine that allows for the incremental delivery of entries during the insertion phase, followed by a phase where the 'smallest' entries are extracted one at a time (where order is determined by the aforementioned predicate).

From a design perspective, this approach is motivated primarily by functional and performance flexibility considerations. In terms of functional flexibility, one clear use case arises when clients only need to order some proper subset of the items previously added (as opposed to an entire collection). And since the component makes no assumptions about the source or destination of the sorted entries, it can serve as a general means of realizing large-effect sorting operations on a variety of other data structures via enhancements. For example, a `Prioritizer` facility can be instantiated in the implementation of a `Sort_Queue` or `Sort_List` enhancement—among other structures.

This design offers performance flexibility as well since the concept interface for the `Prioritizer` merely specifies (abstractly) *what* the operations for insertion, changing phase, and extracting the next 'smallest' entry should accomplish: it's the realization's job to determine *where* and *how* the actual sorting takes place. As a result, performance characteristics can vary substantially across different realizations, so clients must determine which realization best suits their usage situation. The `Prioritizer` example is illustrative of almost every design consideration that motivated the introduction of object-based modularization as an alternative to functional decomposition in [87, 8].

In this chapter, we focus primarily on one particular 'batch' sorting realization that, as the name suggests, simply sorts all inserted entries using a sorting-enhanced queue when the change

phase operation is called. The overall system is summarized in UML in Fig. 6.1 (refer back to Sect. 3.4 for information on UML notational conventions).

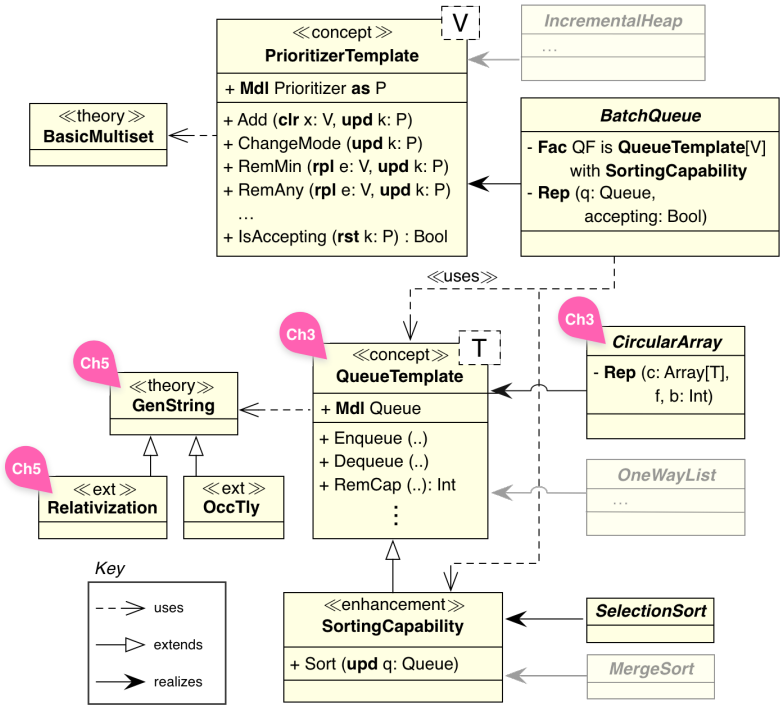


Fig. 6.1: UML for the component-based system developed in this chapter.

Non-faded boxes without markers will be discussed in this chapter while pink tagged boxes indicate the chapter where the particular artifact was discussed. Artifacts in faded/lighter boxes are not directly addressed, though their sources are included in the workspace linked at the very end of this section. Note that all realizations appearing in the diagram rely solely on concept interfaces, as opposed to other realizations. This allows formal reasoning to be performed in a modular fashion: i.e., strictly on the basis of other concepts and their respective mathematical models and specifications.

Verification Challenges.

Though modest in size, the system depicted in Fig. 6.1 presents a challenge to the development of general autoactive specification and verification languages and their corresponding tools. This is largely because our approach does not assume the presence of built-in theories (for inte-

gers, strings, arrays, multisets or otherwise) and thus requires the verifier to reason generally across component boundaries, taking into account multiple user-defined theories.

For example, in the `BatchQueue` realization (Fig. 6.1), verification will necessarily require composition of results from both general string theory, its relativization extension, and a new theory for multisets which we use to model the `Prioritizer`. Our theory of multisets along with an extension for converting from strings to multisets is presented in Sect. 6.2.

This approach stands in contrast to other auto active efforts (such as Dafny [68]) where theories for strings, multisets, arrays, etc. are built-in. Such methods employ efficient (though highly specialized) decision procedures targeting fixed, decidable fragments of a theory's underlying language. See [112] for a look at what goes into the development of one such procedure designed to handle assertions about (imperative style) trees.

And while methods involving decision procedures are efficient, they can quickly become inflexible and slow when combined in nonstandard ways (e.g., when two or more theories happen to make use of same overlapping operators beyond equality) [113, 18]. As it stands, many of the theories and proof obligations we present in this chapter remain somewhat afield of the current capabilities of state-of-the-art solvers because they are either higher order in nature, or require composition of several theories to dispatch.

The sources for the component library developed in this work (including generated proof obligations for the discussed components) are available online at: https://github.com/dtwelch/resolvework.

6.2 An Extension for Fully Generic Sorting

Before we consider the prioritizer component, we turn first to the specification, realization, and verification of a generic sorting enhancement for queues (Fig. 6.2).

RESOLVE Studio

```
Enhancement Sorting_Capability (
          Def ⊴ : Entry × Entry ⟶ 𝔹) for Queue_Template;
  uses Basic_Ordering_Theory;
  requires Is_Total_Preordering(⊴);

  Operation Sort (updates Q : Queue);
      ensures Q Is_Permutation #Q ∧ Is_Cfml_w(Q, ⊴);
end Sorting_Capability;
```

RESOLVE Studio

Fig. 6.2: A generic sorting enhancement for queues.

6.2.1 Abstract Specification and Supporting Predicates

The enhancement is parameterized by an (abstract) predicate $\trianglelefteq$ that determines the ordering for the queue's entries. The module level precondition subsequently `requires` that the relation passed is a total preordering (namely: it must be both total and transitive). The relevant predicate from ordering theory that captures this is given below:

RESOLVE Studio

```
Precis Basic_Ordering_Theory;
    uses Basic_Binary_Relation_Properties, ...;
    Def Is_Total_Preordering (≼ : (D : Cls) × D ⟶ 𝔹) : 𝔹 ≜
                Is_Transitive(≼) ∧ Is_Total(≼);
    // ...
```

RESOLVE Studio

The utility of higher order functions should be readily apparent here. In particular, the total preordering definition takes an arbitrary relation $\preccurlyeq$ as a parameter (which was specialized by $\trianglelefteq$ in the sorting specification) and passes it as an argument to other predicates such as `Is_Total`—which

holds iff $\preccurlyeq$ is a total (i.e., serial) relation:

RESOLVE Studio

```
//Basic_Binary_Relation_Properties.resolve
Def Is_Total (ρ : (D : Cls) × D ⟶ 𝔹) : 𝔹 ≜
        ∀ x, y : D, x ρ y ∨ y ρ x;
```

RESOLVE Studio

The sorting enhancement takes advantage of this formal machinery (in particular, the condition of totality) to guarantee that any two entries are comparable given an arbitrary, user-supplied ordering relation.

To Expand or Not to Expand: A question that naturally arises when considering definitions of this form is how frequently the automated verifier is required to unfold/expand these otherwise "uninterpreted" predicate or function applications during proof search. This question is addressed for several library components and verification benchmarks in [102, 45], where it was found that expansion is generally not required for the vast majority of VCs arising from user programs. The authors do suggest however that this is perhaps less true for concepts involving local definitions operating over specific model types directly (e.g., `o : SomeMdlType` $\longrightarrow T$) or specifications involving heap properties.

The `Sort` operation itself takes a single queue, `Q`, and `ensures` two properties: (i) that the resulting queue is a permutation of the incoming queue's entries (i.e., they consist of exactly the same entries, through are perhaps differently ordered). And (ii), that the ordering of entries in the outgoing queue is conformal with (`Is_Cfml_w`) the $\trianglelefteq$ ordering predicate.

String Conformality. Testing conformality of a string with respect to some binary predicate (though powerful) is the simpler of the two operators, so we define it first:

RESOLVE Studio

```
//Relativization_Ext.resolve
Def Is_Cfml_w (⋉ : (Γ : SSet) × Γ ⟶ 𝔹, β : Str(Γ)) : 𝔹 ≜
   ∀ x, y : Γ, (⟨x⟩ ∘ ⟨y⟩ Is_Substring β) ⟹ x ⋉ y;
```

RESOLVE Studio

The operator takes a binary relation $\ltimes$, a string β and is defined to hold if each pairing of

elements occurring within β satisfy the $\ltimes$ operator passed. Note that because the definition requires quantification over individuals of β (in this case $x, y : \Gamma$), we define it in the relativization extension for strings—as opposed to the general theory, where no assumptions are made about the type of the internal elements. See Appendix A.5.1 for the complete theory, including some corollaries.

String Permutations via Multiset Theory. To facilitate the definition of the aforementioned string permutation predicate, we turn to multiset theory. This also serves as a useful—albeit informal—introduction to the theory (the formal precis for which is given in Sect. 6.3; additional definitions can be found in Appendix A.6).

Multisets are simply unordered collections that permit repeated elements and keep a "tally" of the number of times each element occurs (this is also called the "multiplicity" of an element). The underlying set of a multiset is the set in which each element has a tally of one. The cardinality of a multiset is defined as the sum of the tallies for each element in the underlying set.

Example 3. To help provide a sense of the various multiset (mset) notations and how they function, we consider some concrete examples.

— Multiset Notation Illustration —

```
⟅a, b⟆           // The mset containing a and b
⟅a, b, b, a, b⟆  // The mset containing a with multiplicity 2
                 //       and b with multiplicity 3
⟅ ⟆ = Φ          // The empty mset

// The underlying set of
U_Set(⟅a, a, a, b, b, a, c⟆) = {a, b, c}
U_Set(Φ) = ∅

// M = ⟅a, a, a, b, b, a, c⟆
⌈M, b⌉ = 2 and ⌈M, e⌉ = 0 // M tallied at b and M tallied at e
⟅d, c, c, a, a, a⟆ ∼^M ⟅a, c, c, c, a⟆ = ⟅d, a⟆ // Minus
⟅a, b, a, e⟆ ⊎ ⟅b, e⟆ = ⟅a, a, b, b, e, e⟆      // Union plus

‖⟅a, a, b, b, b, c⟆‖ = 6                // Cardinality of
```

— Multiset Notation Illustration —

We use these operators to define an occurrence tallying function (`Occ_Tly`) that converts

a string to an mset. This function, in turn, will prove useful in the definition of the desired string permutation predicate. Since there are currently no other operators in general string theory that require msets, we confine their definition to a separate extension provided below (recall that `ext(`α`, x)` extends a string α with element `x`):

RESOLVE Studio

```
Precis Occ_Tly_Permute_Ext extends Gen_String_Theory;
   uses Basic_Multiset_Theory;

   Inductive Def Occ_Tly (α : SStr) : MMSet is
     (i.) Occ_Tly(Λ) = Φ
     (ii.) ∀ x : El,
              Occ_Tly(ext(α, x)) = Occ_Tly(α) ⊎ |x|;

    Corollary OT_1: ∀ α, β : SStr,
               Occ_Tly(α ∘ β) = Occ_Tly(α) ⊎ Occ_Tly(β);

    Def (α : SStr) Is_Permutation (β : SStr) : ℙ ≜
               Occ_Tly(α) = Occ_Tly(β);

    //...
end Occ_Tly_Permute_Ext;
```

RESOLVE Studio

With a definition for `Occ_Tly` in place, we can easily express what it means for one string α to be a permutation of β: i.e., by asserting that `Occ_Tly(`α`) = Occ_Tly(`β`)`.

6.2.2 A Selection Sorting Realization

With the necessary mathematics in place, we now discuss a selection sorting realization. Fig. 6.3 shows code realizing the primary `Sort` operation loaded in RESOLVE Studio. The implementation iterates over the queue to be sorted, extracts the minimum entry each iteration via the the local operation `Remove_Min`, then enqueues it onto a temporary queue, `Sorted_Q` (which holds the entries ordered thus far).

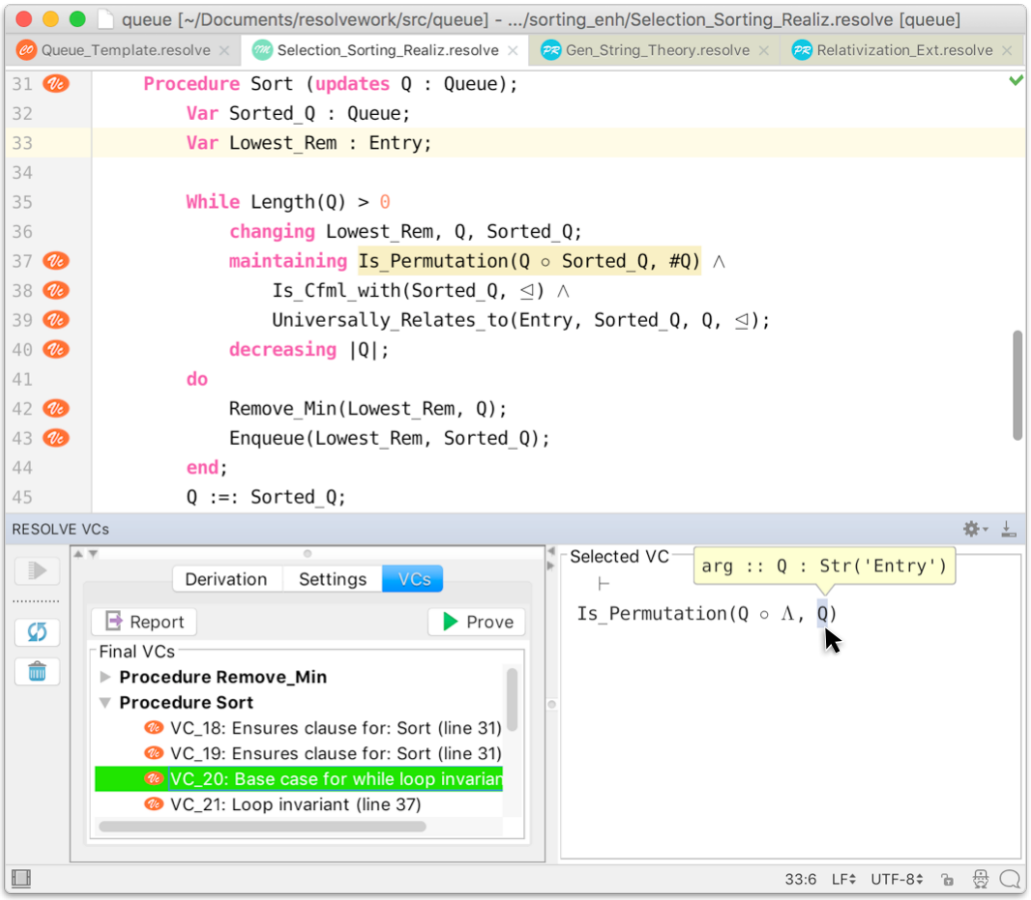


Fig. 6.3: An iterative selection sorting realization in RESOLVE Studio.

The invariant can be summarized as follows:

- The first conjunct states that the concatenation of elements in the (temporary) `Sorted_Q` and `Q` constitute the entirety of the elements being sorted (i.e., that their concatenation is indeed a permutation of the original queue, `#Q`).
- The second conjunct states that `Sorted_Q`'s elements are ordered w.r.t. the $\trianglelefteq$ relation.
- The last conjunct states that every element in `Sorted_Q` is related by $\trianglelefteq$ to every element in `Q` (the call to `Remove_Min` each iteration `ensures` this). In other words, all the entries in `Sorted_Q` "precede" the remaining entries in `Q`.

Clicking one of the  badges opens a menu where users can select any of the VCs arising from that particular line of code. Once clicked, the F-IDE interfaces with the VGui tool and automatically

navigates users to the VC in question, displaying the VC in the lower right-hand window (the region of the sourcefile that generated the goal is also highlighed in the editor). This process is outlined in Fig. 6.4.

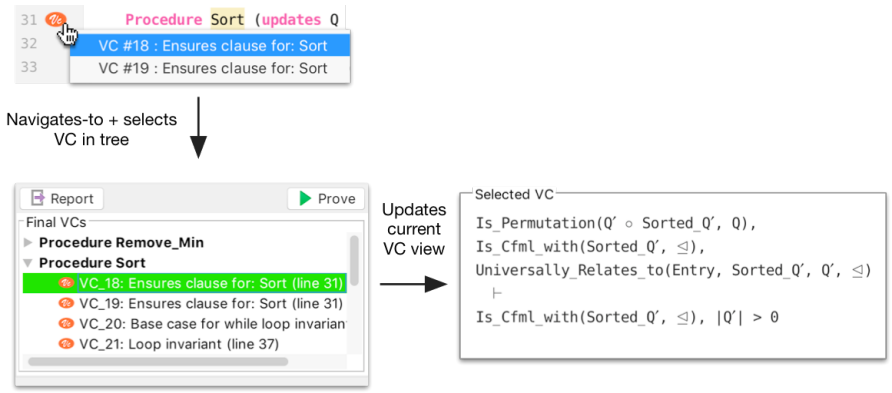


Fig. 6.4: VC selection process in RESOLVE Studio (alternatively, users can select VCs from the tree directly—which has the same effect on the "Selected VC" view); the "Report" button merely writes the generated VCs to file along with their proof status.

Verification Summary.

Most VCs generated from this example are reasonably straightforward (at least from a human perspective). Indeed, after the "prove" button is pressed, most of the trivial VCs (e.g., those involving tautologies or arithmetic bounds) are able to be proven automatically with a timeout of ∼4 seconds. Others however that hinge on subtle algebraic observations (usually unforeseen during the development of a theory) tend to timeout—such as the following VC involving permutations that arises from the loop invariant in `Remove_Min`:

```
((New_Q' ∘ ⟨Min'⟩) ∘ (⟨Temp'⟩ ∘ Q')) Is_Permutation Q,
1 ≤ |⟨Temp'⟩ ∘ Q'|,
|Q| ≠ 0
  ⊢
(((New_Q' ∘ ⟨Temp'⟩) ∘ ⟨Min'⟩) ∘ Q') Is_Permutation Q, Temp' ◁ Min'
```

Clearly the first antecedent should be sufficient to prove this, however the general theory currently lacks suitable results for asserting that concatenation of strings is commutative when applied as an argument to the `Is_Permutation` predicate. Of course, one could always add a lemma to string

theory with the required shape—though these are typically not general enough to warrant inclusion in the theory (especially considering each addition must be proven offline with a proof assistant or otherwise). Development of an automated prover capable of dispatching such assertions—e.g., using a combination of associativity, commutativity, and other general permutation corollaries—remains a topic of ongoing work.

6.3 A (Preliminary) Theory of Multisets

The theory of multisets consists of a class `MMSet`, which denotes the collection of all multisets and thus provides a domain over which mset variables in the theory can range. This is followed by a unary function `U_Set : MMSet` $\longrightarrow$ `SSet` that maps a multiset to its underlying set, a binary function[1] $\lceil \bullet, \bullet \rceil$ `: MMSet` $\times$ `El` $\longrightarrow$ `CCard` where $\lceil M, x \rceil$ tallies a given multiset M at some element x, and—lastly—a binary constructor function:

$\lfloor$`(x : •), •(x)`$\rfloor$ `: ((T : SSet)` $\times$ `(`ψ `: T` $\longrightarrow$ `CCard))` $\longrightarrow$ `MMSet`

that produces the multiset of `x`'s in set `T` with cardinal multiplicity ψ`(x)`.

The core notations (starting with the class universe of all multisets) are formally introduced into the global scope of the theory via a categorical definition:

```
Categorical Def for MMSet : Cls, U_Set : MMSet ⟶ SSet,
   ⌈•,•⌉ : MMSet × El ⟶ CCard,
   ⌊(x : •), •(x)⌋ : (T : SSet) × (ψ : T ⟶ CCard) ⟶ MMSet
 is
   Is_Multiset_like(MMSet, U_Set, ⌈•,•⌉, ⌊(x : •), •(x)⌋);
```

Categorical definitions introduce one or more operators and relate them via the foundational predicate `Is_Multiset_like` (shown below) which encapsulates the essential properties of the theory. Note that—for clarity—the formal parameters to this predicate are given names similar to those introduced above—though each carries a prime to distinguish it syntactically from its global (categorically-) introduced counterpart:

[1] Note that • here is merely used as a placeholder for the position of some formal parameter occurring within the name portion of an outfix operator; this is primarily so we can declare the operator name in a prefix style for consistency

```
Def Is_Multiset_like (MMSet' : Cls, U_Set' : MMSet' ⟶ SSet,
    ⌈•,•⌉' : MMSet' × El ⟶ CCard, ... ) : HB ≜ (
 Pty 1: ∀ M : MMSet', ∀ x : El,
     x ∈ U_Set'(M) ⟺ ⌈M, x⌉' ≠ 0,

 Pty 2: ∀ T : SSet, ∀ ψ : T ⟶ CCard,
         U_Set'( ⌊(x : T), ψ(x)⌋' ) = {x : T | ψ(x) ≠ 0},

 Pty 3: ∀ T : SSet, ∀ ψ : T ⟶ CCard,
     ∀ y : T, ⌈⌊(x : T), ψ(x)⌋', y⌉' = ψ(y),

 Pty 4: ∀ M, N : MMSet', U_Set'(M) = U_Set'(N) ∧
      (∀ x : U_Set'(M), ⌈M, x⌉' = ⌈N, x⌉') ⟹ M = N    );
```

The first property states that if an element x appears in some multiset M's underlying set, then M tallied at x is nonzero (and vice versa). Property two states that the underlying set of the multiset constructor with cardinal multiplicity $\psi(x)$ is equivalent to the set comprehension of elements x : T that occur at least once. Property three states that each element y of mset domain T when tallied maps to its cardinal multiplicity, while property four provides a form of mset extensionality (i.e., that two msets are equal if their underlying sets are equal and share the same number of each element).

The definitions of several other important mset operators are given below in RESOLVE Studio (Fig. 6.5).

```
14      // The empty multiset
15      Literal Def Φ : MMSet ≜ ⌊∅, 0⌋ ;
16          Corollary MSet_Em_1: U_Set(Φ) = ∅;
17          Corollary MSet_Em_2: ∀ M : MMSet, U_Set(M) = ∅ ⟹ M = U
18
19      // The singleton multiset
20      Def ⌊(z : El)⌋ : MMSet ≜ ⌊{z}, Card_Suc(0)⌋ ;
21          Corollary M_S1: ∀ z : El, U_Set( ⌊z⌋ ) = {z};
22          Corollary M_S2: ∀ z : El, ⌈ ⌊z⌋ , z⌉ = 1;
23
24      Def (M : MMSet) ⊎ (N : MMSet) : MMSet;
25          Corollary M_U1: ∀ M, N : MMSet, M ⊎ N = N ⊎ M;
26          Corollary M_U2: ∀ L, M, N : MMSet, L ⊎ (M ⊎ N) = (L ⊎ M) ⊎ N;
27
28      // Bounded multisets over
29      Def BMSet(S : SSet, κ : CCard) : 𝒫(MMSet) ≜
30              {M : MMSet | U_Set(M) : ℘(S) ∧ (∀ x : S, ⌈M, x⌉ ≤ κ)};
31          Corollary M_BM1: ∀ S : SSet, BMSet(S, 0) = {Φ};

    U_Set : MMSet ⟶ SSet
    Card_Suc : CCard ⟶ CCard
    Is_Multiset_like : Cls × MSet ⟶ SSet
    ^↓ and ^↑ will move caret down and up in the editor >>
```

Fig. 6.5: A snippet of multiset theory formalized in RESOLVE Studio.

Among the symbols introduced are the empty multiset Φ, the singleton multiset $\lfloor \bullet \rfloor$, the

multiset union plus operator $\uplus$, and multiset cardinality $\|\bullet\|$. The last definition shown in Fig. 6.5 introduces the notion of a bounded multiset `BMSet`, wherein the number of occurrences of each element within a given set `S` are bounded by the provided cardinal, κ. We can then use this to describe the class of finite multisets (`FMSet`) over a given set `S` as follows:

```
Def FMSet(S : SSet) : ℘(BMSet(S, ℵ_0)) ≜
                                {M : BMSet(S, ℵ_0) | ‖M‖ < ℵ_0};
```

By bounding the cardinality of `M` to be strictly less than $\aleph_0$ (the "smallest" cardinal number), we exclude those sets that are denumerable, hence restricting ourselves to exclusively finite sets.[2]

Suffice to say, for most current program specification purposes (as is reflected in the next section's specifications) finite multisets are generally sufficient. However, if one were to write specifications for a real-time application involving the real numbers $\mathbb{R}$, it is conceivable one would require broader definitions capable of operating over $\aleph_1$, etc.

Multiset Type Checking Concerns.

Type checking multiset theory is a nontrivial task. One source of difficulty is in formulating the definition of `BMSet` and making RESOLVE's type checker aware that `BMSet(S, κ)`—defined to be some subclass of `MMSet`—is "small enough" to inhabit `SSet`, the proper class of all sets. This is achieved with the following recognition:

```
Recognition BMSet_in_SSet:
      ∀ S : SSet,
      ∀ κ : CCard, BMSet(S, κ) : SSet;
```

Such a result proved necessary to ensure the result type for the definition of `FMSet` is type conformal—as the $\wp$ operator requires an argument known to be within `SSet` to satisfy its domain type.

[2] Recall that a set is countable iff its cardinality is finite or equal to $\aleph_0$, a set is denumerable iff its cardinality is exactly $\aleph_0$, and a set is uncountable iff its cardinality is greater than $\aleph_0$. So the empty set is countable, the finite set $\{a,b,c\}$ is countable (but *not* denumerable), the infinite set $\mathbb{N}$ is countable and denumerable, and the set $\mathbb{R}$ is uncountable

Additionally, to preempt any issues in typechecking assertions involving finite msets in the context of the prioritizer template's specification, we introduce the following useful recognitions:

```
Recognition All_FMSets_in_MMSet:
     ∀ S : SSet, ∀ ξ : FMSet(T), ξ : MMSet;

Recognition FMSet_Nat:
      ∀ S : SSet, ∀ M : FMSet(S), ‖M‖ : ℕ;

Recognition FMSet_Tally_Nat:
      ∀ S : SSet, ∀ M : FMSet(S), ∀ x : S, ⌈M, x⌉ : ℕ;
```

The first allows any finite multiset-typed term ξ to be recognized as within `MMSet`, while the second two allow the cardinality and tallying operators to be restricted to the naturals when applied over finite multisets.

6.4 A Prioritizer Concept and A Component-Based Realization

We now have the formal machinery in place to specify the prioritizing component and its queue-based realization overviewed in Sect. 6.1.

6.4.1 Abstract Specification

To start, we create a new project prioritizer that houses the component, its realization, and any client code—then we add the concept shown below in Fig. 6.6.

Like the queue sorting enhancement, the concept is parameterized by a generic type `Label`, a `Max_Capacity`, and a total preordering relation $\preccurlyeq$ that determines entry ordering. The prioritizer type is modeled as the cartesian product of a finite multiset of inserted labels and a boolean flag indicating whether the machine is currently accepting new labels:

`( (Keeper : FMSet(Entry)) × (Is_Accepting : 𝔹) )`.

The `constraints` assert that the cardinality of `K.Keeper` should fall within the maximum capacity of the machine. There are two recognitions at work here. The first, `All_FMSets_in_MMSet`, allows `K.Keeper : FMSet(Entry)` to be applied as an argument to the mset cardinality function—which,

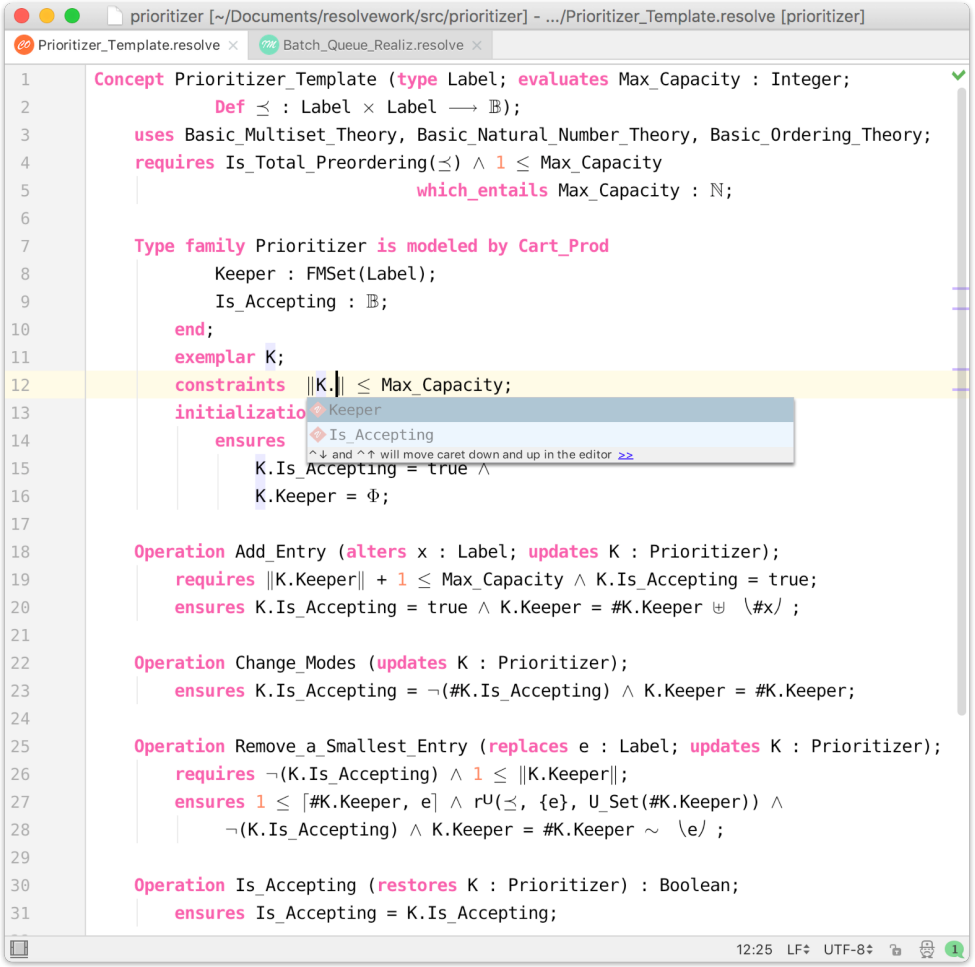


Fig. 6.6: A template for prioritizing generic entries.

by default, is typed as a function from `MMSet` into the cardinals. The second recognition, `FMSet_Nat` from Sect. 6.3, in turn allows ||`K.Keeper`|| to be passed as an argument to natural number theory's ≤ operator.

The `initialization` clause that follows merely `ensures` that the machine starts in an accepting state and that its multiset is initially empty. As outlined in Sect. 6.1, the concept is organized around several "small effect" operations that allow users to: (i) add new labels (assuming there is enough room and the machine is in an accepting state), (ii) switch the state of the machine, and (iii) remove a smallest label. The concept also allows users to retrieve the number of objects stored, remove arbitrary ones, and check the state of the machine.

The postcondition for the remove smallest entry operation warrants some explanation. In addition to utilizing yet another recognition—`FMSet_Tally_Nat` (Sect. 6.3) to typecheck the mset tally application in the first conjunct, it also utilizes a universal relation predicate, $\mathtt{r}^{\mathsf{U}}$, to state that every element in the singleton `{e}` is related to every element in the incoming multiset's underlying set via the $\preccurlyeq$ predicate. This allows us to avoid the introduction of a universal quantifier.

6.4.2 A Sortable Queue-Based Realization

Fig. 6.7 provides a sortable queue-based realization of the prioritizer concept. The realization represents the prioritizer as a record containing an `Items` queue for holding the contents of the prioritizer and a boolean flag, `Accepting`, which tracks the state of the machine.

Note that the queue type, drawn from the `QF` facility, is enhanced with `Sorting_Capability` which is realized by the previously discussed selection sorting realization (Sect 6.2). The facility is instantiated with both the `Label` type and the total capacity bound of the prioritizer. The sorting enhancement-realization pair that follows is specialized by both the abstract $\preccurlyeq$ ordering relation from the concept and the `Labels_are_Ordered` operation which programmatically carries out the $\preccurlyeq$ comparison. The "`from` queue" clause merely tells the compiler which project on RESOLVEPATH contains the subsequently referenced modules.

The `conventions` clause captures a representation invariant asserting that the length of the queue must remain within bounds and that the machine's entries are fully ordered w.r.t. $\preccurlyeq$ whenever the machine is in a non-accepting state. The `correspondence` clause that follows relates the conceptual (`Conc`) state of each model field to a concrete representation variable. In this case, the mapping is functional: the programmatic `Accepting` flag is mapped directly to the conceptual boolean, `Is_Accepting`, while the queue is mapped to the `Keeper` multiset. To abstract the items in the queue we use the `Occ_Tly` function presented in Sect. 6.2.

The realization shown is a not only relatively simple (as most procedures can be implemented using queue operations), but also efficient (as expensive sorting takes place only when `Change_Modes` is called[3]). Note that since the specifications in the concept do not prescribe when

[3]efficiency could be further improved by wrapping the call to `Sort` in a conditional that checks whether the queue as

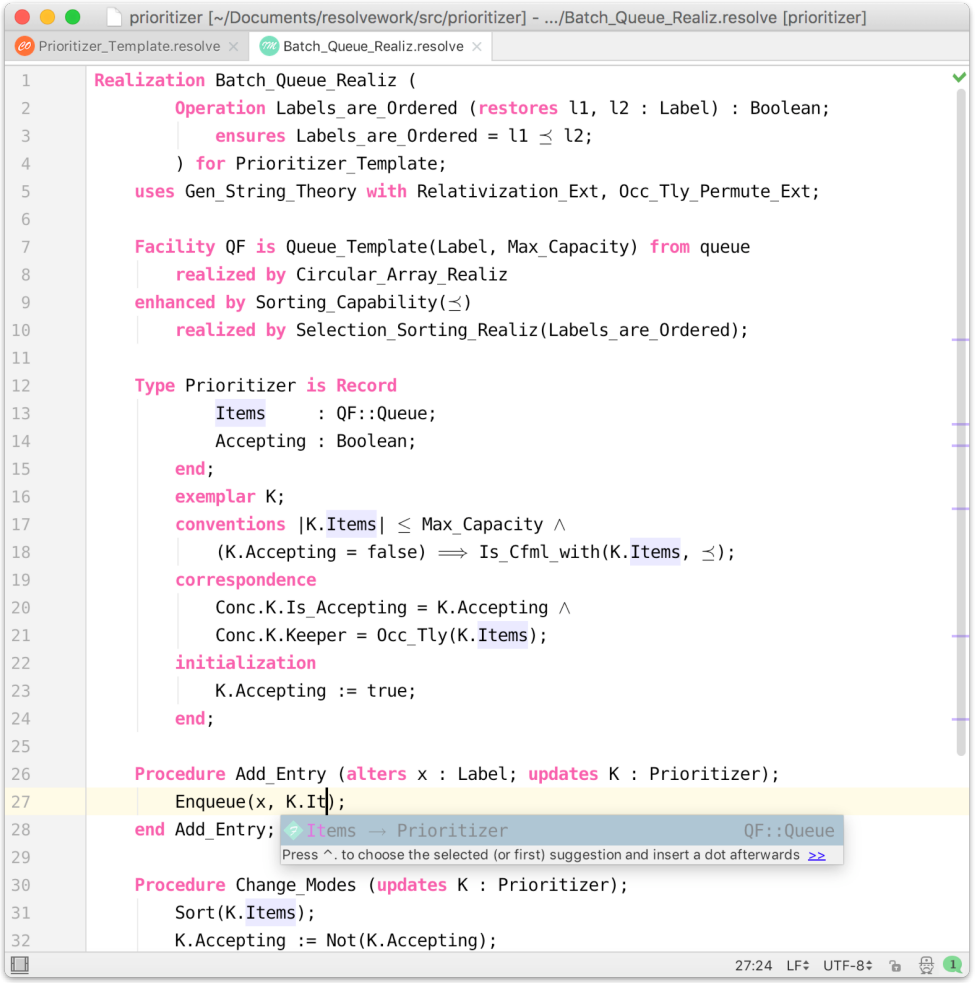


Fig. 6.7: A queue-based realization.

the sorting takes place, one can easily imagine alternative realizations with different performance characteristics and usage considerations. For example, a heap-based realization would immediately add each entry to the heap—and as such, the representation invariant would state that the heap's ordering property is always conformal with $\preccurlyeq$ (see Appendix B.3.3) for work involving one such realization). Another realization might drop the ordering convention altogether and simply find and extract the smallest label each time, as and when needed.

changed since the last invocation of `Change_Modes`

6.4.3 Verification

In this section we survey a handful of VCs that arise from the batch queue realization. Our discussion of the VCs is grouped into sections corresponding to the construct that produced them. For several, we introduce additional corollaries to help make the realization more amenable to automated proof.

Facility Instantiation VCs.

Verification of a facility declaration such as `QF` aims to show that the arguments passed meet the specifications. Processing facility declarations is non-trivial in general. Indeed, before VC generation can even occur, the `QF` facility is subjected to a number of syntactic and static semantic checks. Some include:

- Checking that the number of arguments and formal parameters match for each argument specialized module.
- Finding and propagating instantiations of any generic types appearing in the module's signature into the remaining formal parameters—which may either explicitly or implicitly reference the generic type being replaced. For instance: the queue concept's `Entry` type is replaced by prioritizer's generic `Label`. This mapping is then used to replace all references to `Entry` in the type signatures of $\trianglelefteq$ and `Is_Ordered` (the instantiations are also propagated down into the postcondition term for `Is_Ordered`). Once instantiated, the compiler can proceed to check the actuals against the formals.
- Ensuring that the modes on the parameters of any passed operations are valid for the modes specified in their formal counterparts (this can be statically checked; see col. 3 in Table. **??**).

After ensuring a facility declaration is syntactically well-formed, VCs are generated to establish that the preconditions for each module. The rule also generates VCs establishing that the postcondition of any passed operation, A, is logically strong enough to serve in place of its formal counterpart operation F's argument-specialized postcondition:

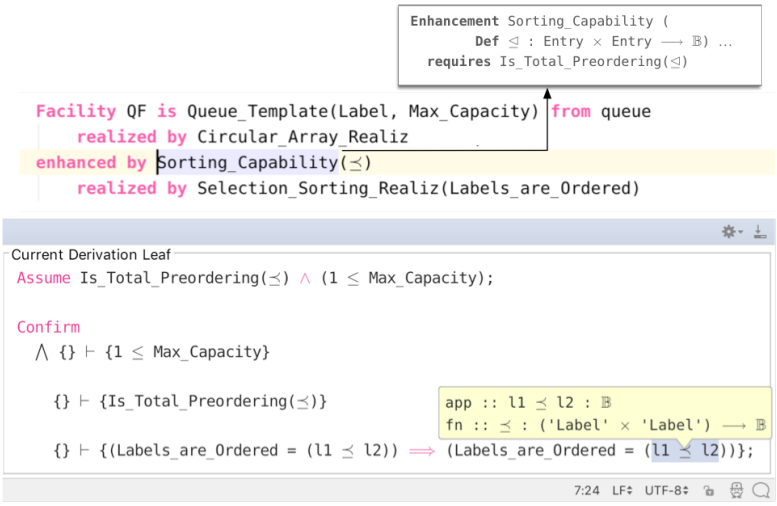


Fig. 6.8: VCs for the `QF` facility pre-simplification and post formal-actual substitution; note that the types for the terms in each formal assertion have been fully instantiated by the prioritizer's generic `Label` type.

$$Post_A \implies Post_F .$$

The opposite must hold true for the precondition. That is, the precondition of the (argument-specialized) formal operation must be at least as strong as the precondition of the actual:

$$Pre_F \implies Pre_A$$

The actual VCs that arise from `QF` are mostly trivial to establish. For example, one requires us to prove `Is_Total_Preordering(≼)` (as per the precondition of `Sorting_Capability`) while another requires us to verify that precondition for `Queue_Template`. The others—as mentioned—involve proofs that the specifications of passed operations are suitable for their formal counterparts. In this case, since the passed operation and its formal (`Is_Ordered`) lack preconditions, the resulting VC is simply $\vdash$ `true` $\implies$ `true` (which is simple enough to be proven outright).

The origins of the VCs resulting from the `QF` facility are visible in RESOLVE Studio's derivation view in Fig. 6.8. Applying the Assume rule to the first statement and ImpRight to the third sequent's succedent results in three VCs, all of which are easily provable since each has an antecedent that is alpha equivalent ($\equiv_\alpha$) to its succeedent.

Type Representation VCs.

Verifying the correctness of a type representation (in the absence of global variables) involves two parts. First, establishing that the type initialization (between the `initialization` and `end` keywords) satisfies the `initialization ensures` clause specified with the model's type in the concept. The VCs generated from the initialization block are the following:

(1) $\vdash$ `Occ_Tly(`Λ`) = `Φ (2) `true = false` $\vdash$ `Is_Cfml_w(`Λ`, `$\preccurlyeq$`)`

(3) `1` $\leq$ `Max_Capacity` $\vdash$ `|`Λ`|` $\leq$ `Max_Capacity`

Most of these follow directly from the definitions of the operators appearing in their succeedents. For example, (1) and (3) follow from the base case of `Occ_Tally` and the string length $|\bullet|$ inductive definitions (respectively). The remaining VC, (3), contains a false assumption and is therefore vacuously true during initialization (this is primarily because the `Accepting` flag is initialized to `true`).

The second part of the rule for generating VCs from type representations involves verifying that the representation's `correspondence` is well defined by establishing that it satisfies the (abstract) `constraints` placed on the model:

`1` $\leq$ `Max_Capacity,`
`|K.Items|` $\leq$ `Max_Capacity` $\vdash$ $\|$`Occ_Tly(K.Items)`$\|$ $\leq$ `Max_Capacity`

To prove this, we add the following corollary to the occurrence tallying theory extension developed in Sect. 6.2:

`Corollary OT_2:` $\forall$ α `: SStr, |`α`|` = $\|$`Occ_Tly(`α`)`$\|$.

The corollary relates the length of a string α to the cardinality of α's occurrence tally multiset. Adding this corollary to the occurrence tally theory extension allows this VC to be verified. Note that, in general, well-designed theories should include useful results such as this by default—the programmer should not be responsible for adding them.

Procedure Level VCs.

Verification of procedures involves generating VCs to ensure that the code satisfies the postcondition of the operation being implemented and that the `conventions` of any involved representation types hold after the code. The following is a representative VC for establishing the postcondition of `Add_Entry`:

```
⊢ Occ_Tly(K.Items ∘ ⟨x⟩) = Occ_Tly(K.Items) ⊎ ⌊x⌋
```

Another interesting VC arises when verifying the conventions clause after `Change_Modes`:

```
K'.Items Is_Permutation K.Items,
Is_Cfml_w(K'.Items, ≼),
|K.Items| ≤ Max_Capacity
  ⊢
|K'.Items| ≤ Max_Capacity
```

Here, since `K'.Items` and `K.Items` are permutations of each other (as per the first antecedent), then their lengths are the same. This effectively connects the third antecedent to the goal, thus proving the VC.

Though the procedures are short, each produces roughly ten VCs. This is due in part to the implication in the `conventions` clause, which—by the proof rule for procedures—is added as an antecedent and split by various applications of the ImpLeft rule—hence producing more (usually trivial) VCs.

Many of the VCs verify outright with only a few requiring the addition of new corollaries (such as the one involving permutations shown above). Fig. 6.9 shows the proof pane in RESOLVE Studio while processing the VCs for the `Change_Modes` procedure.

6.5 Small Case Study: OS-Task Scheduling

In this section we give a small, simplified example to demonstrate how the components designed and presented in this chapter can be used for prioritizing OS task objects. Our example is based roughly on the FreeRTOS [1] API framework—a real-time operating system kernel designed for use on embedded hardware and other micro-controller platforms.

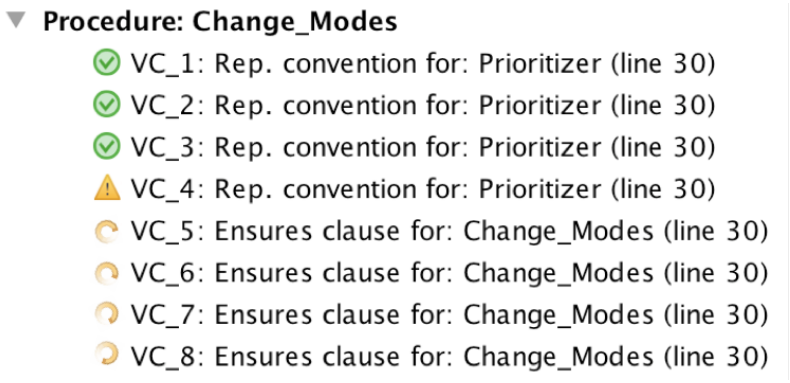


Fig. 6.9: Running the prover on realization VCs

First, we represent a task (or, thread) programmatically as a record consisting of three integer-valued fields: an identifier denoting the process that will be invoked by the scheduler (namely, the FreeRTOS scheduler), the delay of the process in seconds, and its priority with respect to other tasks:

```
Type Task is Record
      T_ID : Integer;
      T_Prio, T_Delay : Integer;
    end;
    exemplar T;
    conventions 0 ≤ T.T_Prio ∧ 0 ≤ T.T_Delay
                which_entails T.T_Prio, T.T_Delay : ℕ;
```

Next, we define a binary predicate on `Task` objects that holds iff task `t2`'s priority is higher than (or equal to) `t1`'s:

```
Def (t1 : Task) ≪^P (t2 : Task) : 𝔹 ≜ t1.T_Prio ≤ t2.T_Prio;
```

alongside the accompanying programmatic implementation of the predicate:

```
Oper Priority_Over (preserves T1, T2 : Task) : Boolean;
    ensures Priority_Over = T1 ≪^P T2;
  Procedure ...
```

With both the datatype and ordering predicates defined, we can instantiate the prioritizer concept as follows:

```
Facility TP is Prioritizer_Template (Task, 3, ≪^P)
  realized by Batch_Queue_Realiz (Priority_Over);
```

Now we define an operation, `Create_Task`, which both initializes the fields of a given task `T`, and adds it to the given task prioritizer `P`:

```
Operation Create_Task (updates P : Prioritizer;
        updates T: Task; evaluates Prio, Dly : Integer);
    requires 0 ≤ Prio ∧ 0 ≤ Dly ∧ ||P.Keeper|| ≤ 3 ∧
        P.Is_Accepting = true;
    ensures T.T_Prio = Prio ∧ T.T_Delay = Dly ∧
            P.Keeper = #P.Keeper ⊎ ⟨T⟩ ∧
            P.Is_Accepting = #P.Is_Accepting;
  Procedure
    T.T_Prio := Prio
    T.T_Delay := Dly;
    TP::Add_Entry(T, P);
end Create_Task;
```

This simplifies the actual FreeRTOS API in the following respects. First, we explicitly move the delay for any invoked task directly into the the task record rather than utilizing function pointers to operations that call, e.g, `Delay_Task`. Moreover, the prioritizer itself would normally be hidden within a separate 'priority' scheduling component, which could maintain and provide additional information on the currently running task as well as those that are delayed, blocked, for example.

A main driver that creates several tasks and retrieves each for scheduling based on its priority is given below along with its sample execution in RESOLVE Studio (Fig. 6.10)

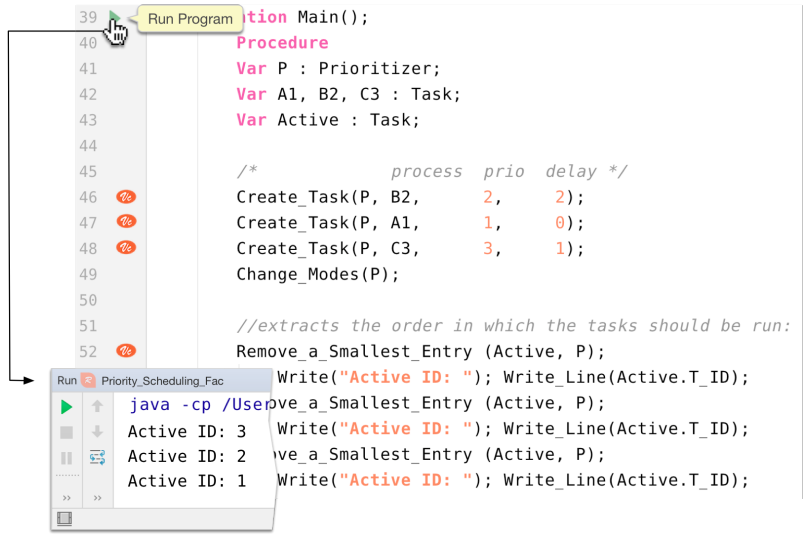


Fig. 6.10: Running the task scheduling driver

Here, we create three tasks `A1`, `B2`, and `C3` where the higher the number suffixing the task denotes a higher priority. One can think of the call to `Change_Modes` as invoking our simplified scheduler. The prioritizing scheduler would first run task `C3`—which immediately gets delayed by a single tick. At this point, task `B2` would preempt this, itself get delayed for two ticks, while `A1` executes.

6.6 Educational Usage

To test the usability of RESOLVE Studio's mathematical editing features and the compiler's new type system, we created an assignment for the roughly dozen students enrolled in Clemson's Spring 2018 graduate programming languages course (CPSC 828). The assignment was structured into two parts.

In the first part, we tasked students with installing RESOLVE Studio, the underlying compiler, then using it to construct a minimal 'toy' version of string theory. Students were asked to complete the following:

1. Create a theory, `Basic_Strings`, and add uninterpreted definitions for the empty string Λ, string extension `ext`, length $|\bullet|$, and singleton $\langle\bullet\rangle$.[4]

2. Formulate an injectivity axiom for `ext` asserting that two strings are equal if the members comprising their extensions are equal.

3. Add an inductive definition to the theory called `Occ_Set` that takes a string α and converts its contents to a set. Then construct two (true) corollaries involving this new definition.

4. Create another theory called `Perfect` that includes a predicate `Is_Perfect_Sq` that holds iff the provided natural number `n` is a perfect square.

5. Add a small theorem to the theory (`Perfect`) stating that the negative integers cannot be perfect squares.

[4]Note that for this and proceeding questions, the signatures of non-nullary operators were provided to students along with an informal description of their purpose

The second half of the assignment had students reproduce the theory in the Coq proof assistant.

6.6.1 Assignment Setup

To prepare students for the first part, setup instructions for the toolchain were provided, along with a basic introduction to RESOLVE's specification language, theory modules, and its type system. Students were shown how to define new symbols (including those with function types), and how to establish subtype relationships through recognitions. Aspects of the F-IDE were also demonstrated: e.g., the reference completion engine and features for inserting non-ASCII characters into the editor.

For the second half of the assignment, students were provided links to various reference materials on Coq, its type system, and its libraries. Since Coq is a relatively popular and well established proof assistant, such materials were readily available online.

6.6.2 Experiences and Observations

Observations on Part 1.

Students reported a generally positive impression of the first part of the assignment—the main stumbling blocks being difficulties in formulating inductive definitions, as well as a general unfamiliarity with RESOLVE's syntax. In various one-on-one help sessions with students, it was observed that they relied frequently on the F-IDE's keyword and reference completion suggestions as well as compiler-provided feedback appearing as annotations in the editor.

One question that proved particularly difficult involved formulating the inductive `Occ_Set` operator (the skeleton syntax for which was given in advance):

RESOLVE Studio

```
Inductive Def Occ_Set(α : SStr) : SSet is
  (i) Occ_Set(Λ) = ∅;
  (ii.) ∀ x, Occ_Set(ext(α, x)) = Occ_Set(α) ∪ {x};
```

RESOLVE Studio

While few had difficulties formulating the base case `(i)`, a significant number (excepting one or two with a background in functional programming) had difficulty in writing the inductive case: `(ii)`.

Another question that caused some difficulty was #5. In answering this, students were required to apply an integer-typed variable to the `Is_Perfect_Sq` predicate (which was defined to take an $\mathbb{N}$). As a result, this particular question required the creation of a new *conditional* recognition asserting that all non-negative integers *i* can also be considered of type natural:

$\forall$ `i` : $\mathbb{Z}$, `i` $\geq$ `0` $\implies$ `i` : $\mathbb{N}$.

Other miscellaneous difficulties are enumerated below:

- Since RESOLVE Studio does not (yet) support Windows, several students had to use Linux or MacOS-based machines to complete the assignment.
- To many, it was unclear which operators were available for use ($\cup$, $\times$, etc.). We observed that most turned to the F-IDE to assist with this. For example, some used a combination of goto declaration to navigate across imported files in search of symbols (without needing to rummage through a filesystem), while others would simply query the active scope to bring up a contextual completion menu containing a list of the available operators (and their types) by pressing `Ctrl` + `Space`.
- In the earlier builds of RESOLVE Studio used for this assignment, students would occasionally receive unintuitive compiler error messages. This was perhaps most significant when dealing with type mismatch errors that occurred when attempting to supply an argument of type $\mathbb{B}$ to class theory based junctors defined over the meta boolean type, `HB`. Foundational pitfalls such as these were understandably confusing. This has since been addressed by allowing the compiler to implicitly convert between the two.

Observations on Part 2.

The results of the second half of the assignment were more variable. The main difficulties came from several sources. First, Coq relies on a foundational theory of types known as the calculus

of inductive constructions (CIC) [25]. Subtyping under Coq's implementation of CIC is handled much differently—resembling more closely the notion of type classes found in Haskell and a variety of other functional programming languages.

Second, Coq's libraries are modeled quite differently than RESOLVE's. For instance, sets in Coq are polymorphic in the type of their entries and rely heavily on a number of different type classes—some of which carry additional constraints when used with new student-defined datatypes.

Lastly, the primary editing environment for Coq is Emacs. Most students in the course had little experience editing in this style, so some opted to use different editors such as VS Code with community-built Coq extensions (though it was found that these generally support fewer features than Coq's main Emacs environment).

Overall it took students some additional effort (and a lot of reading) to form and compose the various definitions needed while also satisfying Coq's type checker.

6.6.3 Summary

The experience suggests that all formal systems will require suitable resources for beginning formal methods users. In particular, we observed that RESOLVE Studio's various features such as completions and hover text (for diagnosing errors—type or otherwise) are nearly essential for ease of use.

Chapter 7

Conclusions and Future Work

7.1 Summary and Conclusions

This dissertation examines how to facilitate design and use of component-based systems. Specifically, we have examined how this can be made possible through the use of an F-IDE designed to minimize the extrinsic difficulties when confronted with the intrinsically difficult task of formal specification and realization design.

To this end, this work presented several new features of a compiler for an integrated specification and programming language that provides users with (i) type-level feedback when constructing mathematical specifications via a prototype type checking scheme, and (ii) a general purpose verification frontend that integrates with the compiler's VC generator and its existing automated prover. The added math type system enables users to flexibility describe subtype relationships between terms of different classifications, while the VGui frontend provides advanced users with the ability to interactively derive VCs and apply (built-in) simplification rules to make resulting VCs smaller and (potentially) more amenable to automation.

The research combines these features into a standalone Formalization Integrated Development Environment (F-IDE) for RESOLVE that provide's many of the amenities that users of modern IDEs have come to expect. This includes reference completions for specifications and code, intention actions to help users avoid errors in specification design, and integration of a code generator

that generates Java code from formally verified components that can be executed in a single click from within the F-IDE.

The F-IDE is then used to conduct a case study where we formalize a generic selection sorting enhancement for queues and utilize it in the realization of a general prioritizing concept. A new theory for multisets was also introduced to help simplify the design of the specifications of the prioritizing concept, along with additional mathematical developments designed to relate and compose new multiset operators with RESOLVE's existing mathematical unit for strings. The VCs that arise through various component interactions are also examined, along with a small case study whereby we use the developed application to specify, simulate, and execute a simple scheduler for OS task objects. While the resulting VCs are sufficiently small and straightforward, some of the more intricate ones (e.g., those involving algebraic properties such as permutations) remain somewhat afield of the current verifier and are among candidate topics for future work.

7.2 Future Work

There are a number of avenues for future work. Some directions include further development of concrete tools such as the VGui frontend (Chap. 4) and RESOLVE Studio (Chap. 5), while others entail new case studies and further development of the RESOLVE language and its compiler. Each item listed is preceded by a bracketed name denoting which part of the system the work being described would target.

- **[F-IDE] Intention Actions.** As noted at the end of Chap. 5, an immediate direction for future work involves the creation of new intention actions for RESOLVE Studio. These would target and provide automatic fixes for a class of common syntactic pitfalls (e.g., misuse of parameter modes) that are frequently overlooked by novice users. These can be added to the F-IDE incrementally, and would improve the responsiveness and power of the tool.

- **[VGui] Theory Simplification Rules.** The design of the VGui verification frontend supports application of general, theory-based rewrite rules to simplify VCs during the VC generation process. Future work would include a suitable syntax and semantics for such rules. This

could ultimately help yield simpler, more human-comprehensible VCs for components with operations that have large and/or complex specifications.

- **[VGui] "Edu" Mode.** While the current setup of the VGui tool is tailored more towards advanced users, future work could also add a mode to the tool that hides all the intermediate steps and derives final VCs for annotation in the IDE. This mode would make several subtle adjustments to the GUI itself and could be activated through a menu.

- **[Compiler, F-IDE] Proof Caching and Annotation.** Another useful aspect of tool feedback includes the ability to cache/persistently store verification results from a given component. This information could be maintained by the compiler and would, in turn, enable useful forms of markup within the F-IDE. For example, the verification status of operations, modules, or even entire projects on RESOLVEPATH could be annotated within the F-IDE's editor, library browser, and various completion menus. Such an addition would visually indicate to clients whether or not they are building upon "trusted" (i.e., previously-verified) code.

- **[Compiler, General] Type Theory Foundations.** Further work is needed to fully formalize the foundational aspects of RESOLVE's type checker. This includes a full formalization of the system's elaborated first order foundations as well as a full formalization of rules for context formation and classification checking.

- **[General] Larger Case Studies.** Future work should also naturally consider larger component-based systems whose specifications and realizations cut across multiple theories. Additional work is also needed to determine how well an automated prover can cope with VCs involving extensive use of other higher order operators and assertions (such as those involving lambda abstractions and set comprehensions). Some of this work is restricted at the moment by limitations of our own verifier in addition to other state-of-the-art SMT solvers such Z3 and CVC4 (which are still subject to first-order restrictions).

- **[Education] Experiments with Students and Researchers.** Ultimately we need multiple field experiments to fully evaluate the existing features and enhance them. While experiments

with graduate and undergraduate students will help us better understand beginning student needs, experiments with researchers will inform us better of the needs of sophisticated users.

Appendices

Appendix A Math Theories

To keep this appendix of a reasonable length, we provide in the following theories only the definitions used directly in this work (and some other closely related operators). For each definition, we also include several important corollaries.

A.1 Basic Binary Operation Properties

```
Precis Basic_Binary_Op_Properties;
    Def Is_Associative (⊙ : (D : SSet) × D ⟶ D) : 𝔹 ≜
        ∀ x, y, z : D, x ⊙ (y ⊙ z) = (x ⊙ y) ⊙ z;

    Def Is_Commutative (⊙ : (D : SSet) × D ⟶ D) : 𝔹 ≜
         ∀ x : D, Is_Commutator_for(⊙, x);

    Def Is_Commutator_for (⊙ : (D : SSet) × D ⟶ D, c : D) : 𝔹 ≜
         ∀ y : D, c ⊙ y = y ⊙ c;

    Def Is_Right_Identity_for (⊙ : (D : SSet) × D ⟶ D, i : D) : 𝔹 ≜
        (∀ x : D, x ⊙ i = x);

    Def Is_Left_Identity_for (⊙ : (D : SSet) × D ⟶ D, j : D) : 𝔹 ≜
        (∀ x : D, j ⊙ x = x);

    Def Is_Identity_for (⊙ : (D : SSet) × D ⟶ D, i : D) : 𝔹 ≜
        (Is_Right_Identity_for(⊙, i) ∧ Is_Left_Identity_for(⊙, i));

    Theorem Op1: ∀ D : SSet, ∀ ⊙ : D × D ⟶ D, ∀ i, j : D,
        Is_Right_Identity_for(⊙, i) ∧
        Is_Left_Identity_for(⊙, j) ⟹ i = j;

end Basic_Binary_Op_Properties;
```

A.2 Basic Binary Relation Properties

```
Precis Basic_Binary_Reln_Properties;
    Def Is_Reflexive (γ : (D : SSet) × D ⟶ 𝔹) : 𝔹 ≜
        ∀ x : D, x γ x;
```

```
    Def Is_Symmetric (γ : (D : SSet) × D ⟶ 𝔹) : 𝔹 ≜
        ∀ x, y : D, x γ y ⟹ y γ x;

    Def Is_Transitive (γ : (D : SSet) × D ⟶ 𝔹) : 𝔹 ≜
        ∀ x, y, z : D, x γ y ∧ y γ z ⟹ x γ z;

    Def Is_Total (γ : (D : SSet) × D ⟶ 𝔹) : 𝔹 ≜
       ∀ x, y : D, x γ y ∨ y γ z;

end Basic_Binary_Reln_Properties;
```

A.3 Basic Ordering Theory

```
Precis Basic_Ordering_Theory;
    uses Basic_Binary_Reln_Properties;

    Def Is_Preordering (⊴ : (D : SSet) × D ⟶ 𝔹) : 𝔹 ≜
        Is_Reflexive(⊴) ∧ Is_Total(⊴);

    Def Is_Total_Preordering (⊴ : (D : SSet) × D ⟶ 𝔹) : 𝔹 ≜
        Is_Transitive(⊴) ∧ Is_Total(⊴)

    // Determines if, for each pairing of elements between sets S1 and S2,
    // the provided predicate ⊴ holds
    Def r^U (⊴ : (T : SSet) × T ⟶ 𝔹, S1 : ℘(T), S2 : ℘(T)) : 𝔹 ≜
      ∀ x : S1, ∀ y : S2, x ⊴ y;

end Basic_Ordering_Theory;
```

A.4 Function Theory

```
Precis Function_Theory;
    Def Is_Identity (f : (D : SSet) ⟶ D) : 𝔹 ≜ (∀ x : D, f(x) = x);

    Def Is_Injective (f : (D : SSet) ⟶ (R : SSet)) : 𝔹 ≜
        (∀ x, y : D, f(x) = f(y) ⟹ x = y);
```

```
    Def Is_Surjective (f : (D : SSet) ⟶ (R : SSet)) : 𝔹 ≜
        ∀ y : R, ∃ x : D, f(x) = y;

    Def Is_Bijective (f : (D : SSet) ⟶ (R : SSet)) : 𝔹 ≜
        Is_Injective(f) ∧ Is_Surjective(f);

    //The ordered pairing operator ⟨•,•⟩ is defined at the meta level
    //in Class_Theory as is FFn, the class of all functions
    Def Dom (F : FFn) : Cls ≜ {y : El | ∃ x : El, ⟨x, y⟩ : F}

    Def Im (F : FFn) : Cls ≜ {y : El | ∃ x : El, ⟨x, y⟩ : F}

    //Function composition
    Def Im (F : FFn) ∘ (G : FFn) : Cls;

    //Iterated function application: IA(f, x, 3) ≡ f(f(f(x)))
    Def IA (f : (D : SSet) ⟶ D, s : D, n : ℕ) : D;

end Function_Theory;
```

A.4.1 Set Application Extension

```
Precis Set_App_Op_Ext extends Function_Theory;

  Def App (f : (T : SSet) ⟶ (U : SSet), S : ℘(T)) : ℘(U) ≜
        {u : U | ∃ x : S, f(x) = u};

    Corollary App_C1: ∀ D, R : SSet, ∀ f : D ⟶ R, App(f, ∅) = ∅;

    Corollary App_C2: ∀ D, R : SSet, ∀ f : D ⟶ R, App(f, D) = Im(f);

    Corollary App_C3: ∀ D, R : SSet,
        ∀ f : D ⟶ R,
            Is_Surjective(f) ⟹ App(f, D) = R;

    Corollary App_C4: ∀ D, R : SSet,
        ∀ f : D ⟶ R,
        ∀ S, T : ℘(D),
            S ⊆ T ⟹ App(f, S) ⊆ App(f, T);
```

```
    Corollary App_C5: ∀ D, R : SSet,
        ∀ f : D ⟶ R,
        ∀ S, T : ℘(D),
            App(f, S ∪ T) = App(f, S) ∪ App(f, T);

end Set_App_Op_Ext;
```

A.5 General String Theory

```
Precis General_String_Theory;
    uses Basic_Natural_Number_Theory;

    //𝒫 is "Powerclass" (defined in Class_Theory)
    Def Is_String_Former (SStr' : Cls, Λ' : SStr',
                          ext' : SStr' × El ⟹ SStr') : HB ≜ (
       Pty 1 (empty string): ∀ α : SStr', ∀ x : El, ext'(α, x) ≠ Λ';
       Pty 2 (extensionality): ∀ α, β : SStr', ∀ x, y : El,
                       ext'(α, x) = ext'(β, y) ⟹ α = β ∧ x = y;
       Pty 3 (well-founded induction): ∀ S : 𝒫(SStr'),
               (  Λ' ∈ S ∧
                  (∀ α : S, ∀ x : El, ext'(α, x) ∈ C) ) ⟹ SStr' = C;
    );

    Categorical Def for
                 SStr : Cls,
                 Λ : SStr,
                 ext : SStr × El ⟶ SStr is
        is_String_Former(SStr, Λ, ext);

    Inductive Def (α : SStr) ∘ (β : SStr) : SStr is
        (i.) α ∘ Λ = α;
        (ii.) ∀ x : El, α ∘ ext(β, x) = ext(α ∘ β, x);

      Corollary C1: Is_Identity_for(∘, Λ);
      Corollary C2: Is_Associative(∘);

    Inductive Def |(α : SStr)| : ℕ is
        (i.) |Λ| = 0;
        (ii.) ∀ x : El, α ∘ ext(β, x) = ext(α ∘ β, x);

      Corollary L1: ∀ α : SStr, |α| = 0 ⟺ α = Λ;
      Corollary L2: ∀ α, β : SStr, |α ∘ β| = |α| + |β|;
```

```
    Def ⟨ (x : El) ⟩ : SStr ≜ ext(Λ, x);
      Corollary C_C1: ∀ x : El, ⟨x⟩ ≠ Λ;
      Corollary C_C2: ∀ x : El, |⟨x⟩| = 1;
      Corollary C_C3: ∀ α : SStr, ∀ x : El, α ∘ ⟨x⟩ = ext(α, x);
      Corollary C_C4: Is_Injective("⟨•⟩");

    Inductive Def Rev(α : SStr) : SStr is
        (i.) Rev(Λ) = Λ;
        (ii.) ∀ x : El, Rev(ext(α, x)) = ⟨x⟩ ∘ Rev(α);

      Corollary R_C1: ∀ x : El, Rev(⟨x⟩) = ⟨x⟩;
      Corollary R_C2: ∀ α, β : SStr, Rev(α ∘ β) = Rev(β) ∘ Rev(α);
      Corollary R_C3: ∀ α ; SStr, Rev(α) = α;
      Corollary R_C4: ∀ α : SStr, |Rev(α)| = |α|;

    Def (α : SStr) Is_Substring (β : SStr) : 𝔹;
      Corollary IS_1: ∀ α, β : SStr,
              α Is_Substring (α ∘ β) ∧ β Is_Substring (α ∘ β);
      Corollary IS_2: ∀ α, β : SStr,
              α Is_Substring β ⟹ |α| ≤ |β|;

    Inductive Def Prt_Btwn (m, n : ℕ, α : SStr) : SStr is
        (i.) Prt_Btwn(m, n, Λ) = Λ;
        (ii.) ∀ x : El, Prt_Btwn(m, n, ext(α, x)) =
                if |α| < n then ext(Prt_Btwn(m, n, α), x)
                else Prt_Btwn(m, n, α);
      Corollary PB_1: ∀ α : SStr, ∀ n : ℕ, n ≥ |α| ⟹
                  (Prt_Btwn(0, n, α) = α)
      Corollary PB_2: ∀ α : SStr, ∀ n : ℕ, Prt_Btwn(n, n, α) = Λ;

end General_String_Theory;
```

A.5.1 String Relativization Extension

```
Precis Relativization_Ext extends General_String_Theory;

    //Constructs strings of a homogeneous type, Γ
    Def Str (Γ : Cls) : 𝒫(SStr) ≜ {α : SStr | Occ_Set(α) ⊆ Γ};
      Corollary S_1: Str(El) = SStr;
```

```
    Recognition Empty_Str_in_all_Strs:
        ∀ Γ : Cls, Λ : Str(Γ);
    Recognition Stringleton_in_all_Strs:
        ∀ Γ : Cls, ∀ x : Γ, ⟨x⟩ : Str(Γ);

  Def Universally_Relates_to (
        α : Str(Γ : SSet), β : Str(Γ),
        ⋉ : Γ × Γ ⟶ 𝔹) : 𝔹 ≜
      (∀ x, y : Γ, ⟨x⟩ Is_Substring α ∧
                   ⟨y⟩ Is_Substring β ⟹ x ⋉ y);

    Corollary URt_1: ∀ Γ : SSet, ∀ ⋉ : Γ × Γ ⟶ 𝔹, ∀ α : Str(Γ),
        Universally_Relates_to(α, Λ, ⋉) ∧
        Universally_Relates_to(Λ, α, ⋉);

    Corollary URt_2:
        ∀ Γ : SSet, ∀ ⋉ : Γ × Γ ⟶ 𝔹, ∀ x, y : Γ,
            Universally_Relates_to(⟨x⟩, ⟨y⟩, ⋉) ⟺ x ⋉ y;

    Corollary URt_3:
        ∀ Γ : SSet,
        ∀ ⋉ : Γ × Γ ⟶ 𝔹,
        ∀ α, β : Str(Γ),
           (Is_Cfml_w(⋉, α) ∧ Is_Cfml_w(⋉, β) ∧
            Universally_Relates_to(α, β, ⋉)
                                ⟹ Is_Cfml_w(⋉, α ∘ β);

  Def Is_Cfml_w (β : Str(Γ : SSet), ⋉ : Γ × Γ ⟶ 𝔹) : 𝔹 ≜
      (∀ x, y : Γ, (⟨x⟩ ∘ ⟨y⟩ Is_Substring β) ⟹ (x ⋉ y));

    Corollary ICw_1: ∀ Γ : SSet, ∀ ⋉ : Γ × Γ ⟶ 𝔹,
       Is_Cfml_w(⋉, Λ) ∧
           (∀ x, y : Γ, Is_Cfml(⋉, ⟨x⟩) ∧
                            (x ⋉ y ⟺ Is_Cfml_w(⋉, ⟨x⟩ ∘ ⟨y⟩)));
end Relativization_Ext;
```

A.5.2 String Occurrence Tally and Permutation Extension

```
Precis Occ_Tly_Permute_Ext extends General_String_Theory;
    uses Basic_Multiset_Theory;
```

```
     // Occurrence Tally
    Inductive Def Occ_Tly (α : SStr) : MMSet is
        (i.) Occ_Tly(Λ) = Φ;
        (ii.) ∀ x : El, Occ_Tly(ext(α, x)) = Occ_Tly(α) ⊎ ⌊x⌋ ;

      Corollary OT_1: ∀ α, β : SStr,
              Occ_Tly(α ∘ β) = Occ_Tly(α) ⊎ Occ_Tly(β);

      Corollary OT_2: ∀ α : SStr, |α| = ‖Occ_Tly(α)‖;
      Corollary OT_3: ∀ α : SStr, Occ_Set(α) = U_Set(Occ_Tly(α));
      Corollary OT_4: ∀ α : SStr, Occ_Tly(Rev(α)) = Occ_Tly(α);

    Def (α : SStr) Is_Permutation (β : SStr) : 𝔹 ≜
                                    Occ_Tly(α) = Occ_Tly(β);
      Corollary P_1: ∀ α, β : SStr,
              (α ∘ β) Is_Permutation (β ∘ α);
      Corollary P_2: ∀ α, β : SStr,
              (α Is_Permutation β) ⟹ (|α| = |β|);
      Corollary P_3: ∀ α, β : SStr,
              (α Is_Permutation β) ⟹ Occ_Set(α) = Occ_Set(β);
end Occ_Tly_Permute_Ext;
```

A.6 Basic Multiset Theory

Note that `Basic_Cardinal_Theory` at the moment is quite small: it exports a class universe `CCard`, along with (uninterpreted) operators for $\aleph_0$: `CCard` and the base element `0` : `CCard`. Further development of this theory remains a topic for future work.

```
Precis Basic_Multiset_Theory;
    uses Basic_Cardinal_Number_Theory,
         Basic_Natural_Number_Theory;

    Def Is_Multiset_like (MMSet' : Cls, U_Set' : MMSet' ⟶ SSet,
            ⌈•,•⌉' : MMSet' × El ⟶ CCard,
            ⌊(x : •), •(x)⌋' : (T : SSet) × (ψ : T ⟶ CCard) ⟶ MMSet'
        ) : HB ≜ (
       Pty 1 (uset membership): ∀ M : MMSet', ∀ x : El,
             x ∈ U_Set'(M) ⟺ ⌈M, x⌉' ≠ 0,

       Pty 2 (nonzero multiplicity): ∀ T : SSet, ∀ ψ : T ⟶ CCard,
             U_Set'( ⌊(x : T), ψ(x)⌋' ) = {x : T | ψ(x) ≠ 0},
```

```
   Pty 3 (cardinal multiplicity): ∀ T : SSet, ∀ ψ : T ⟶ CCard,
         ∀ y : T, ⌈⌊(x : T), ψ(x)⌋', y⌉' = ψ(y),

   Pty 4 (extensionality): ∀ M, N : MMSet', U_Set'(M) = U_Set'(N) ∧
         (∀ x : U_Set'(M), ⌈M, x⌉' = ⌈N, x⌉') ⟹ M = N
);

Categorical Def for MMSet : Cls, U_Set : MMSet ⟶ SSet,
    ⌈•,•⌉ : MMSet × El ⟶ CCard,
    ⌊(x : •), •(x)⌋ : (T : SSet) × (ψ : T ⟶ CCard) ⟶ MMSet
  is
    Is_Multiset_like(MMSet, U_Set, "⌈•,•⌉", "⌊•,•⌋");

Literal Def Φ : MMSet ≜ ⌊∅, 0⌋;
  Corollary EM_1: U_Set(Φ) = ∅;
  Corollary EM_2: ∀ M : MMSet, (U_Set(M) = ∅) ⟹ (M = Φ);

Def ⌊(z : El)⌋ : MMSet ≜ ⌊{z}, Card_Suc(0)⌋;
  Corollary MU_1: ∀ M, N : MMSet, M ⊎ N = N ⊎ M;
  Corollary MU_2: ∀ L, M, N : MMSet,
         ∀ X : SSet, L ⊎ (M ⊎ N) = (L ⊎ M) ⊎ N;
  Corollary MU_3: ∀ M, N : MMSet,
         U_Set(M ⊎ N) = U_Set(M) ∪ U_Set(N);

// Bounded multisets over
Def BMSet (S : SSet, κ : CCard) : 𝒫(MMSet) ≜
        {M : MMSet | U_Set(M) : ℘(S) ∧ (∀ x : S, ⌈M, x⌉ ≤ κ)};
   Corollary BMS_1: ∀ S : SSet, BMSet(S, 0) = {Φ};
   Corollary BMS_2: ∀ κ : CCard, BMSet(∅, κ) = {Φ};

Def ‖(M : MMSet)‖ : CCard ≜ Sum (x : U_Set(M), ⌈M, x⌉);

// Uniformely tallied to (produces a multiset where each element in
// set S occurs κ times in the resulting multiset)
Def (S : SSet) ⋆ (κ : CCard) : MMSet ≜ ⌊S, κ⌋;

// Application over multisets
Def App^M ( f: (D: Set) ⟶ (R: Set), M : MMSet) : MMSet;
  Corollary MA_1: ∀ D, R : SSet, ∀ f : D ⟶ R, ∀ M : MMSet,
     U_Set(App^M(f, M)) ⊆ R;

// Finite multisets
Def FMSet(S : SSet) : SSet ≜ {M : BMSet(S, ℵ_0) | ‖M‖ < ℵ_0};
```

```
    Recognition All_FMSets_in_MMSet:
        ∀ S : SSet, ∀ ξ : FMSet(T), ξ : MMSet;

    Recognition FMSet_Nat:
        ∀ S : SSet, ∀ M : FMSet(S), ‖M‖ : ℕ;

    Recognition FMSet_Tally_Nat:
       ∀ S : SSet, ∀ M : FMSet(S), ∀ x : S, ⌈M, x⌉ : ℕ;
end Basic_Multiset_Theory;
```

A.7 Basic k-Spiral Theory

This is a work-in-progress theory designed to facilitate reasoning about a type of data structure termed a *k*-spiral (discussed further in Appendix B.3.2). The theory utilizes a form of dependent types, as the value *k* is bound in and appears throughout type-level terms (i.e., those on the right hand side of a colon). While the theory shown does indeed typecheck under our current (semi-formal) typing rules, further research is needed fully formalize how values appearing in types are treated—and perhaps impose certain syntactic restrictions on their use.

```
Precis Basic_Spiral_Theory;
    uses Basic_Natural_Number_Theory;

    Def ℕ₂ : ℘(ℕ) ≜ {n : ℕ | n ≥ 2};
      Recognition All_N2_in_N: ∀ n : ℕ₂, n : ℕ;

    Def Is_k_Spiral_Like(Sp_Loc' : ℕ₂ ⟶ SSet,
              Cen' : (k : ℕ₂) ⟶ Sp_Loc'(k),
              SS', RS' : (k : ℕ₂) ⟶ Sp_Loc'(k) ⟶ Sp_Loc'(k)
    ) : HB ≜ (
      Pty 1 (successor reachability): ∀ k : ℕ₂, ∀ p : Sp_Loc'(k),
               ∃ n : ℕ, IA(SS'(k), Cen'(k), n) = p;
      Pty 2 (successor injectivity): ∀ k : ℕ₂, ∀ m, n : ℕ,
               (IA(SS'(k), Cen'(k), m) = IA(SS'(k), Cen'(k), n)) ⟹ m = n;
      Pty 3 (central-radial successor identity):
               ∀ k : ℕ₂, RS'(k)(Cen'(k)) = SS'(k)(Cen'(k));
      Pty 4 (radial successor arity):
               ∀ k : ℕ₂, ∀ p : Sp_Loc(k),
                  RS'(k)(SS'(k)(p)) = IA(SS'(k), Cen'(k), k);
    );
```

```
Categorical Def for Sp_Loc : ℕ₂ ⟶ SSet,
    SS, RS : (k : ℕ₂) ⟶ Sp_Loc(k) ⟶ Sp_Loc(k),
    Cen : (k : ℕ₂) ⟶ Sp_Lock(k)
  is
     Is_k_Spiral_Like(Sp_Loc, Cen, SS, RS);

// Spiral center distance (SCD):
Implicit Def SCD : (k : ℕ₂) ⟶ (p : Sp_Loc(k)) ⟶ ℕ is
    ∀ p : Sp_Loc(k), IA(SS(k), Cen(k), SCD(k)(p)) = p;
  Corollary SCD_1: ∀ k : ℕ₂,
          ∀ p : Sp_Loc(k), SCD(k)(p) < SCD(k)(RS(k)(p));
  Corollary SCD_2: ∀ k : ℕ₂,
          ∀ p : Sp_Loc(k), SS(k)(p) = IA(SS(k), Cen(k), SCD(k)(p) + 1);
  Corollary SCD_3: ∀ k : ℕ₂, ∀ n : ℕ, SCD(k)(IA(SS(k), Cen(k), n) = n;

// Radial predecessor
Def RP : (k : ℕ₂) ⟶ ( (p : Sp_Loc(k)) ⟶ Sp_Loc(k) );

// Spiral offset distance
Def SOD : (k : ℕ₂) ⟶ ( (p : Sp_Loc(k)) ⟶ ℕ );
  Corollary R_1: ∀ k : ℕ₂, ∀ p : Sp_Loc(k), ∀ i : ℕ,
   i < k ⟹ p = RP(k)( IA(SS(k), RS(k)(p), i) ) ∧
             i = SOD(k)( IA(SS(k), RS(k)(RP(k)(p)), i) )

// In sector of
Def (q : Sp_Loc(k : ℕ₂)) In_Sect_of (p : Sp_Loc(k)) : 𝔹 ≜
               ∃ n : ℕ, IA(RP(k), q, n) = p;
  Corollary ISo_1: ∀ k : ℕ₂, ∀ p, q, r : Sp_Loc(k),
          r In_Sect_of p ∧
          r In_Sect_of q ⟹ (p In_Sect_of q ∨ q In_Sect_of p);

Def (q : Sp_Loc(k : ℕ₂)) Is_Inside_of (p : Sp_Loc(k)) : 𝔹 ≜
              ∃ n : ℕ, IA(SS(k), q, n) = p);

    Corollary IIo_1: ∀ k : ℕ₂, ∀ q, p : Sp_Loc(k),
              q Is_Inside_of p ⟹ SCD(k)(q) ≤ SCD(k)(p);
    Corollary IIo_2: ∀ k : ℕ₂, ∀ q, p : Sp_Loc(k),
              q Is_Sect_of p ⟹ p Is_Inside_of q;

Def Inward_Loc(p : Sp_Loc(k : ℕ₂)) : ℘(Sp_Loc(k)) ≜
         {q : Sp_Loc(k) | SCD(k)(q) < SCD(k)(p)};

    Corollary IL_1: ∀ k : ℕ₂, ∀ b : Sp_Loc(k),
              Cen(k) ∈ Inward_Loc(SS(k)(b));
```

```
    Corollary IL_2:  ∀ k : ℕ₂, ∀ m, n : ℕ,
              IA(SS(k), Cen(k), m) ∈ Inward_Loc( IA(SS(k), Cen(k), n) )
                                   ⟹ m < n;
    Corollary IL_3: ∀ k : ℕ₂, ∀ b, p : Sp_Loc(k),
              b In_Sect_of p ∧ b ≠ p ⟹ p ∈ Inward_Loc(b);
    Corollary IL_4: ∀ k : ℕ₂, ∀ b, p : Sp_Loc(k),
              p ∈ Inward_Loc(b) ⟹ Inward_Loc(p) ⊂ Inward_Loc(b);
    Corollary IL_5: ∀ k : ℕ₂, ∀ p, q : Sp_Loc(k),
              Inward_Loc(p) ⊆ Inward_Loc(q) ∨
              Inward_Loc(q) ⊆ Inward_Loc(p);

// Node labels in sector bounded by b and p ordered w.r.t. ⋉
Def Is_Sect_Cfml_for (F : Sp_Loc(k : ℕ₂) ⟶ (Γ : SSet),
                      ⋉ : Γ × Γ ⟶ 𝔹, b, p : Sp_Loc(k)) : 𝔹 ≜
     ∀ q : Sp_Loc(k), SS(k)(q) Is_Inside_of b ∧
              RP(k)(q) In_Sect_of p ⟹ F(q) ⋉ F(RP(k)(q));

  Corollary ISCf_1: ∀ k : ℕ₂, ∀ Γ : SSet, ∀ f : Sp_Loc(k) ⟶ Γ,
      ∀ ⋉ : Γ × Γ ⟶ 𝔹,
      ∀ b, p : Sp_Loc(k),
          b Is_Inside_of RS(k)(p) ⟹ Is_Sect_Cfml_for(f, ⋉, b, p);
```

Appendix B Concepts, Enhancements, and Realizations

B.1 Queue Template

```
Concept Queue_Template (type Entry; evaluates Max_Length : Integer)
    uses General_String_Theory with Relativization_Ext,
         Basic_Natural_Number_Theory;
    requires 1 ≤ Max_Length which_entails Max_Length : ℕ;

    Type family Queue is modeled by Str(Entry);
        exemplar Q;
        constraints |Q| ≤ Max_Length;
        initialization ensures Q = Λ;

    Operation Enqueue (alters e : Entry; updates Q : Queue);
        requires 1 + |Q| ≤ Max_Length;
        ensures Q = #Q ∘ ⟨#e⟩;

    Operation Dequeue (replaces e : Entry; updates Q : Queue);
        requires 1 ≤ |Q|;
        ensures #Q = ⟨e⟩ ∘ Q;

    Operation Swap_First_Entry (updates e : Entry; updates Q : Queue);
        requires 1 ≤ |Q|;
        ensures e = DeString(Prt_Btwn(0, 1, #Q)) ∧
                  Q = ⟨#e⟩ ∘ Prt_Btwn(1, |#Q|, #Q);

    Operation Length (restores Q : Queue) : Integer;
        ensures Length = |Q|;

    Operation Rem_Capacity (restores Q : Queue) : Integer;
        ensures Rem_Capacity = (Max_Length - |Q|);

    Operation Clear (clear Q : Queue);
end Queue_Template;
```

B.1.1 Circular Array Realiz

```
Realization Circular_Array_Realiz for Queue_Template;

    Type Queue is Record
```

```
            Contents: Array 0..Max_Length - 1 of Entry;
            Prefront, Length : Integer;
        end;
        conventions
           0 ≤ Q.Prefront < Max_Length ∧ 0 ≤ Q.Length ≤ Max_Length;
        correspondence
           Conc.Q = ∏ (i : Integer, Q.Prefront + 1, Q.Prefront + Q.Length,
                   ⟨Q.Contents(i mod Max_Length)⟩);

    Procedure Enqueue (alters e : Entry; updates Q : Queue);
        Var Temp : Integer;
        Q.Length := Q.Length + 1;
        Temp := Q.Prefront + Q.Length;
        Temp := Temp mod Max_Length;
        Q.Contents[Temp] :=: e;
    end Enqueue;

    Procedure Dequeue (replaces e : Entry; updates Q : Queue);
        Var Temp : Integer;
        Temp := Q.Prefront + 1;
        Q.Prefront := Temp mod Max_Length;
        Q.Contents[Q.Prefront] :=: e;
        Q.Length := Q.Length - 1;
    end Dequeue;

    Procedure Swap_First_Entry (updates e : Entry; updates Q : Queue);
        Var Temp : Integer;
        Temp := Q.Prefront + 1;
        Temp := Temp mod Max_Length;
        Q.Contents[Temp] :=: e;
    end Swap_First_Entry;

    Procedure Length (restores Q : Queue) : Integer;
        Length := Q.Length;
    end Length;

    Procedure Rem_Capacity (restores Q : Queue) : Integer;
        Rem_Capacity := Max_Length - Q.Length;
    end Rem_Capacity;

    Procedure Clear (clears Q : Queue);
       Q.Prefront := 0;
       Q.Length := 0;
    end Clear;
end Circular_Array_Realiz;
```

B.2 Queue Sorting Capability

```
Enhancement Sorting_Capability (
        Def ⊴ : Entry × Entry ⟶ 𝔹) for Queue_Template;
    uses Basic_Ordering_Theory,
         Gen_String_Theory with Occ_Tly_Permute_Ext;
    requires Is_Total_Preordering(⊴);

    Operation Sort (updates Q : Queue);
        ensures Q Is_Permutation #Q ∧ Is_Cfml_w(Q, ⊴);
end Sorting_Capability;
```

B.2.1 Selection Sorting Realiz for Sorting Capability

```
Realization Selection_Sorting_Realiz (
        Operation Is_Ordered (restores e1, e2 : Entry) : Boolean;
                 ensures Is_Ordered = e1 ⊴ e2;
        )
        for Sorting_Capability;

    Operation Remove_Min (replaces Min : Entry; updates Q : Queue);
          requires |Q| ≠ 0;
          ensures (Q ∘ ⟨Min⟩) Is_Permutation #Q ∧
                  Universally_Relates_to(Q, ⟨Min⟩, ⊴) ∧
                  (|Q| = |#Q| - 1);
        Procedure
            Var Temp : Entry;
            Var New_Q : Queue;
            Dequeue(Min, Q);
            While 1 <= Length(Q)
                changing Q, New_Q, Min, Temp;
                maintaining (New_Q ∘ ⟨Min⟩ ∘ Q) Is_Permutation #Q ∧
                    Universally_Relates_to(New_Q, ⟨Min⟩, ⊴);
                decreasing |Q|;
            do
                Dequeue(Temp, Q);
                If Is_Ordered(Temp, Min) then
                    Min :=: Temp;
```

```
                end;
                Enqueue(Temp, New_Q);
            end;
            New_Q :=: Q;
    end Remove_Min;

    Procedure Sort (updates Q : Queue);
        Var Sorted_Q : Queue;
        Var Lowest_Rem : Entry;

        While Length(Q) > 0
            changing Lowest_Rem, Q, Sorted_Q;
            maintaining Is_Permutation(Q ∘ Sorted_Q, #Q) ∧
                Is_Cfml_w(Sorted_Q, ⊴) ∧
                Universally_Relates_to(Sorted_Q, Q, ⊴);
            decreasing |Q|;
        do
            Remove_Min(Lowest_Rem, Q);
            Enqueue(Lowest_Rem, Sorted_Q);
        end;
        Q :=: Sorted_Q;
    end Sort;

end Selection_Sorting_Realiz;
```

B.3 Prioritizer Template

```
Concept Prioritizer_Template (type Label;
              evaluates Max_Capacity : Integer;
              Def ≼ : Label × Label ⟶ 𝔹);
    uses Basic_Multiset_Theory,
         Basic_Natural_Number_Theory, Basic_Ordering_Theory;
    requires Is_Total_Preordering(≼) ∧ 1 ≤ Max_Capacity
                                which_entails Max_Capacity : ℕ;

    Type family Prioritizer is modeled by Cart_Prod
            Keeper : FMSet(Label);
            Is_Accepting : 𝔹;
        end;
        exemplar K;
        constraints ‖K.Keeper‖ ≤ Max_Capacity;
        initialization ensures
```

```
            K.Is_Accepting = true ∧ K.Keeper = Φ;

    Operation Add_Entry (alters x : Label; updates K : Prioritizer);
        requires ‖K.Keeper‖ + 1 ≤ Max_Capacity ∧ K.Is_Accepting = true;
        ensures K.Is_Accepting = true ∧
                K.Keeper = #K.Keeper ⊎ ⟨#x⟩;

    Operation Change_Modes (updates K : Prioritizer);
        ensures K.Is_Accepting = ¬ #K.Is_Accepting ∧
                K.Keeper = #K.Keeper;

    Operation Remove_a_Smallest_Entry (replaces e : Label;
                                       updates K : Prioritizer);
        requires ¬ K.Is_Accepting ∧ 1 ≤ ‖K.Keeper‖;
        ensures 1 ≤ ⌈#K.Keeper, e⌉ ∧ r^u(≼, {e}, U_Set(#K.Keeper)) ∧
             ¬ K.Is_Accepting ∧ K.Keeper = #K.Keeper ~ ⟨e⟩;

    Operation Is_Accepting (restores K : Prioritizer) : Boolean;
        ensures Is_Accepting = K.Is_Accepting;

    Operation Total_Entry_Count (restores K : Prioritizer) : Integer;
        ensures Total_Entry_Count = ‖K.Keeper‖;
end Prioritizer_Template;
```

B.3.1 Batch Queue Realization

This realization uses a sorting-enhanced queue to order the entries whenever `Change_Modes` is called.

```
Realization Batch_Queue_Realiz (
        Operation Labels_are_Ordered (restores l1, l2 : Label) : Boolean;
            ensures Labels_are_Ordered = l1 ≼ l2;
        ) for Prioritizer_Template;
    uses Gen_String_Theory with Relativization_Ext, Occ_Tly_Permute_Ext;

    Facility QF is Queue_Template (Label, Max_Capacity) from queue
        realized by Circular_Array_Realiz
    enhanced by Sorting_Capability (≼)
        realized by Selection_Sorting_Realiz (Labels_are_Ordered);

    Type Prioritizer is Record
```

```
            Items : QF::Queue;
            Accepting : Boolean;
        end;
        exemplar K;
        conventions |K.Items| ≤ Max_Capacity ∧
            (K.Accepting = false) ⟹ Is_Cfml_with(K.Items, ≼);
        correspondence
            Conc.K.Is_Accepting = K.Accepting ∧
            Conc.K.Keeper = Occ_Tly(K.Items);
        initialization
            K.Accepting := true;
        end;

    Procedure Add_Entry (alters x : Label; updates K : Prioritizer);
        Enqueue(x, K.Items);
    end Add_Entry;

    Procedure Change_Modes (updates K : Prioritizer);
        Sort(K.Items);
        K.Accepting := not K.Accepting;
    end Change_Modes;

    Procedure Remove_a_Smallest_Entry (replaces e : Label;
                                       updates K : Prioritizer);
       Dequeue (e, K.Items);
    end Remove_a_Smallest_Entry;

    Procedure Is_Accepting (restores K : Prioritizer);
        Is_Accepting := K.Accepting;
    end Is_Accepting;

    Procedure Total_Entry_Count (restores K : Prioritizer);
        Total_Entry_Count := Length(K.Items);
    end Total_Entry_Count;

end Batch_Queue_Realiz;
```

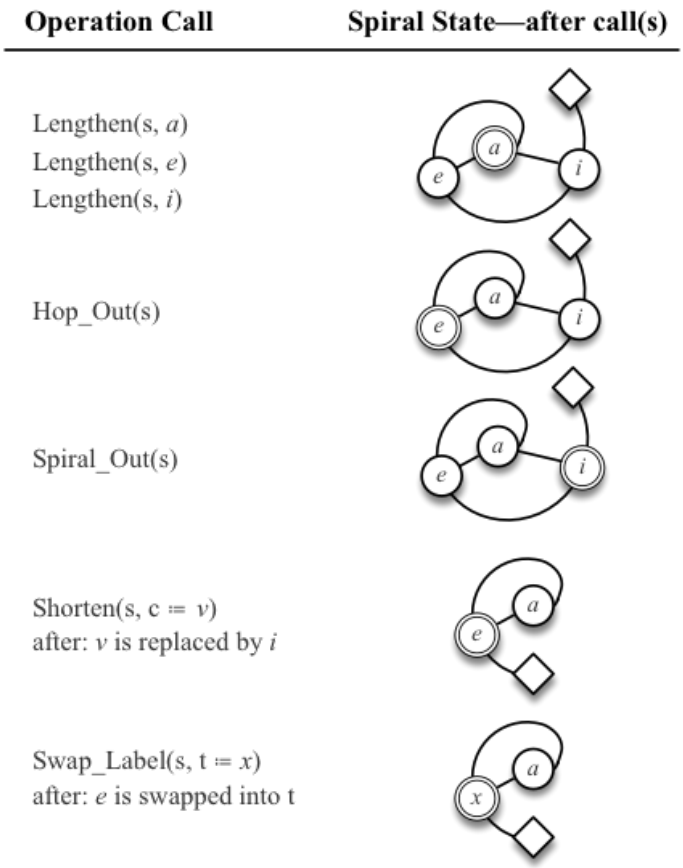


Fig. 1: Spiral concept primary operations. The double circle indicates the current cursor position within the spiral while the diamond shape marks the spiral's terminal location

B.3.2 Spiral Template Fragment

A *k*-spiral is an experimental concept developed as a general structure for storing labels hierarchically. A fragment of the concept, including its preamble and type family, is provided below while Fig. 1 illustrates a subset of its primary operations in action.

```
Concept Spiral_Template (type Label; evaluates k, Max_Length : Integer);
    uses Basic_Spiral_Theory, Basic_Natural_Number_Theory;
    requires 2 ≤ k ∧ 1 ≤ Max_Length
                          which_entails k : N2 ∧ Max_Length : ℕ;

    Type family Spiral_Pos is modeled by Cart_Prod
            Lab : Sp_Loc(k) ⟶ Label;
            Curr_Loc,
            Trmnl_Loc : Sp_Loc(k);
        end;
        exemplar P;
        constraints SCD(k)(P.Curr_Loc) < Max_Length ∧
```

```
            P.Curr_Loc Is_Inside_of P.Trmnl_Loc ∧
            (∀ p : Sp_Loc(k), SCD(k)(p) ≤ SCD(k)(P.Trmnl_Loc) ⟹
                  P.Lab(p) = Label.Base_Point);
        initialization
            ensures P.Trmnl_Loc = Cen(k);
    ...
end Spiral_Template;
```

Labels are stored sequentially along the spiral's arcs and can be traversed linearly or vertically (by hopping up towards the center or down towards the edge of the arcs). The spiral enforces balance, as new locations can only be added to the next available slot on the outermost arc. We conjecture that spirals can be used in certain cases as a somewhat simpler, conceptual alternative to balanced tree-like structures such as red black trees (which require intricate balancing procedures and invariants [32, 77]).

B.3.3 Prioritizer Heap-Based Realization

The following is an alternative realization of the prioritizer concept that uses the *k*-spiral as the representation of the heap in place of what traditionally would be an array. Use of the spiral and its accompanying theory encapsulates the algebraic results necessary to abstractly describe heap properties.

```
Realization Incremental_Heap_Realiz (
        Operation Labels_are_Ordered (restores l1, l2 : Label) : Boolean;
            ensures Labels_are_Ordered = l1 ≼ l2
        ) for Prioritizer_Template;
    uses Basic_Spiral_Theory from spiral;

    /* Note: Under construction */

    Facility Heap_Fac is Spiral_Template
                            (Entry, 2, Max_Capacity) from spiral
       realized by Array_Realiz
     enhanced by Checking_Ops
       realized by Iterative_Checking_Realiz;

    Type Prioritizer is Record
            Heap : Heap_Fac::Spiral_Pos;
            Accpt_Flag : Boolean;
```

```
    end;
    exemplar K;
    conventions
        //The heap ordering property holds via ≼
        Is_Sect_Cfml_for(K.Heap.Lab, ≼, K.Heap.Trmnl_Loc, Cen(2));
    correspondence
        Conc.K.Is_Accepting = K.Accpt_Flag ∧
        Conc.K.Keeper =
              App^M(K.Heap.Lab, Inward_Loc(K.Heap.Trmnl_Loc) ⋆ 1);

Def Is_Relabeling_of (P, Q : Heap_Fac::Spiral_Pos) : 𝔹 ≜
   App^M(P.Lab, Inward_Loc(P.Trmnl_Loc) ⋆ 1) =
                              App^M(Q.Lab, Inward_Loc(Q.Trmnl_Loc) ⋆ 1);

// Note: This is a definition might be a candidate for spiral theory
// (it likely needs several corollaries)
Def Min_Loc_of (P, Q : Sp_Loc(2),
                f : Entry × Entry ⟶ 𝔹) : Sp_Loc(2);

Operation Reposition_Cursor_to_Smallest (updates P : Spiral_Pos);
    requires SS(2)(SS(2)(P.Curr_Loc)) ∈ Inward_Loc(P.Trmnl_Loc);
    ensures P.Curr_Loc =
        Min_Loc_of(SS(2)(#P.Curr_Loc), SS(2)(SS(2)(#P.Curr_Loc)), ≼) ∧
        P.Lab = #P.Lab ∧ P.Trmnl_Loc = #P.Trmnl_Loc;
    Procedure
        Var Left, Right : Entry;
        Var L_Side : Boolean;

        Swap_Label(P, Left);
        Spiral_Out(P);
        Swap_Label(P, Right);

        L_Side := Labels_are_Ordered(Left, Right);

        Swap_Label(P, Right);
        Spiral_In(P);
        Swap_Label(P, Left);
        If L_Side = false then
           Spiral_Out(P);
        end;
end Reposition_Cursor_to_Smallest;

//From the Checking_Ops enhancement:
/*
    Operation At_Center (restores P : Spiral_Pos) : Boolean;
```

```
      ensures At_Center = (SCD(k)(P.Curr_Loc) = 0);

    Operation At_Last_Meaningful_Pos (
                          restores P : Spiral_Pos) : Boolean;
      ensures At_Last_Meaningful_Pos =
                (SS(k)(P.Curr_Loc) = P.Trmnl_Loc);
*/

//Otherwise known as "Heapify"
Operation Fix_Pos (updates P : Spiral_Pos);
    requires
        ∀ q : Sp_Loc(2), RP(2)(q) = P.Curr_Loc ⟹
                Is_Bdd_Sect_Cfml_for(P.Lab, ≼, P.Trmnl_Loc, q);
    ensures P Is_Relabeling_of #P ∧
        Is_Bdd_Sect_Cfml_for(P.Lab, ≼, P.Trmnl_Loc, #P.Curr_Loc);
    Recursive Procedure
        Var Top, Smallest_Sect_Pos : Entry;
        Var Offset_Num : Integer;
        If At_Edge(P) = false then
            Swap_Label(P, Top);
            Hop_Out(P);
            If At_Last_Meaningful_Pos(P) = false then
                 Reposition_Cursor_to_Smallest(P);
            end;
            Swap_Label(P, Smallest_Sect_Pos);

            If Labels_are_Ordered(Smallest_Sect_Pos, Top) then
                Top :=: Smallest_Sect_Pos;
            end;

            Swap_Label(P, Smallest_Sect_Pos);
            Fix_Pos(P);
            Hop_In(P, Offset_Num);
            Swap_Label(P, Top);
        end;
end Fix_Pos;

Procedure Add_Entry (alters x : Label; updates K : Prioritizer);
    Var Offset_Num : Integer;
    Move_to_End(K.Heap);
    Lengthen(K.Heap, x);

    While At_Center(K.Heap) /= true
        changing K.Heap, Offset_Num;
        maintaining
```

```
                    Is_Bdd_Sect_Cfml_for(K.Heap.Lab, ≼,
                            K.Heap.Trmnl_Loc, #K.Heap.Curr_Loc) ∧
                (K.Heap Is_Relabeling_of #K.Heap) ∧
                        (K.Accpt_Flag = #K.Accpt_Flag);
            decreasing SCD(2)(K.Heap.Curr_Loc);
        do
            Hop_In(K.Heap, Offset_Num);
            Fix_Pos(K.Heap);
        end;
    end Add_Entry;

    Procedure Remove_a_Smallest_Entry (replaces s : Entry;
                                       updates K : Prioritizer);
        Shorten(K.Heap, s);
        If Not At_Center(K.Heap) then
            Move_to_Center(K.Heap);
            Swap_Label(K.Heap, s);
            Fix_Pos(K.Heap);
        end;
    end Remove_a_Smallest_Entry;

    Procedure Change_Modes (updates K : Prioritizer);
        K.Accpt_Flag := not K.Accpt_Flag;
    end Change_Modes;

    Procedure Total_Entry_Count (restores K : Prioritizer) : Integer;
        Total_Entry_Count := Length_of(K.Heap);
    end Total_Entry_Count;

    Procedure Is_Accepting_Entries (restores K : Prioritizer) : Boolean;
        Is_Accepting_Entries := K.Accpt_Flag;
    end Is_Accepting_Entries;

    Procedure Clear (clears K : Prioritizer);
        K.Accpt_Flag := true;
        Clear(K.Heap);
    end Clear;
end Incremental_Heap_Realiz;
```

Bibliography

[1] The freertos project. Available on www.freertos.org.

[2] Parosh Aziz Abdulla and K. Rustan M. Leino. Tools for software verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 15(2):85–88, 2013.

[3] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *LNCS*. Springer, Cham, 2016.

[4] Michael Ameri and Carlo A. Furia. Why just Boogie? In Erika Ábrahám and Marieke Huisman, editors, *IFM 2016*, pages 79–95, Cham, 2016. Springer.

[5] Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A formally verified proof of the central limit theorem. *Journal of Automated Reasoning*, 2017.

[6] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV 2011*, volume 6806 of *LNCS*, pages 171–177, Berlin, Heidelberg, 2011. Springer.

[7] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *CAV 2007*, volume 4590 of *LNCS*, pages 298–302, Berlin, Heidelberg, 2007. Springer.

[8] Wolfram Bartussek and David Lorge Parnas. Using assertions about traces to write abstract specifications for software modules. In *ISM*, volume 65 of *LNCS*, pages 211–236. Springer, 1978.

[9] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[10] Dines Bjørner and Klaus Havelund. 40 years of formal methods. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014*, volume 8442 of *LNCS*, pages 42–61, Cham, Switzerland, 2014. Springer.

[11] François Bobot, Sylvain Conchon, Evelyne Contejean, Mohamed Iguernelala, and Stéphane Lescuyer. The Alt-Ergo automated theorem prover. Website, available at http://alt-ergo.lri.fr/.

[12] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3 - sheperd your herd of provers. In K. Rustan M. Leino and Michał Moskal, editors, *Boogie 2011*, pages 53–64, 2011.

[13] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6):709–727, 2015.

[14] Sascha Böhme. *Proving Theorems of Higher-Order Logic with SMT Solvers*. PhD thesis, Technische Universität München, 2012.

[15] Tevfik Bultan and Aysu Betin-Can. Scalable software model checking using design for verification. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE 2005*, volume 4171 of *LNCS*, pages 337–346, Cham, 2005. Springer.

[16] Samuel R. Buss. An introduction to proof theory. *Bulletin of Symbolic Logic*, 6(4):464–465, 2000.

[17] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, Frans M. Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *SOSP 2015*, pages 18–37. ACM, 2015.

[18] Paula Chocron, Pascal Fontaine, and Christophe Ringeissen. Politeness and combination methods for theories with bridging functions. *Journal of Automated Reasoning*, 2019.

[19] Maria Christakis, K. Rustan M. Leino, Peter Müller, and Valentin Wüstholz. Integrated environment for diagnosing verification errors. In Marsha Chechik and Jean-François Raskin, editors, *TACAS 2016*, volume 9636 of *LNCS*, pages 424–441, Berlin, Heidelberg, 2016. Springer.

[20] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs 1981*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[21] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[22] Byron Cook, Heidy Khlaaf, and Nir Piterman. On automation of CTL* verification for infinite-state systems. In Daniel Kroening and Corina S. Pasareanu, editors, *CAV 2015*, volume 9206 of *LNCS*, pages 13–29. Springer, 2015.

[23] Charles T. Cook, Svetlana Drachova-Strang, Yu-Shan Sun, Murali Sitaraman, Jeffrey C. Carver, and Joseph E. Hollingsworth. Specification and reasoning in SE projects using a web IDE. In *CSEE&T 2013, San Francisco, CA, USA, May 19-21, 2013*, pages 229–238. IEEE, 2013.

[24] Charles T. Cook, Heather K. Harton, Hampton Smith, and Murali Sitaraman. Specification engineering and modular verification using a web-integrated verifying compiler. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *ICSE 2012*, pages 1379–1382. IEEE Computer Society, 2012.

[25] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[26] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-end verification of information-flow security for C and assembly programs. In Chandra Krintz and Emery Berger, editors, *PLDI 2016*, pages 648–664. ACM, 2016.

[27] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, 1979.

[28] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[29] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *CADE-25 2015*, volume 9195 of *LNCS*, pages 378–388, Berlin, Heidelberg, 2015. Springer.

[30] Sumesh Divakaran, Deepak D'Souza, Anirudh Kushwah, Prahladavaradan Sampath, Nigamanth Sridhar, and Jim Woodcock. Refinement-based verification of the FreeRTOS scheduler in VCC. In Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *ICFEM 2015*, volume 9407 of *LNCS*, pages 170–186, Berlin, Heidelberg, 2015. Springer.

[31] Svetlana V. Drachova, Jason O. Hallstrom, Joseph E. Hollingsworth, Joan Krone, Rich Pak, and Murali Sitaraman. Teaching mathematical reasoning principles for software correctness and its assessment. *Transactions on Computing Education*, 15(3):15:1–15:22, 2015.

[32] Claire Dross and Yannick Moy. Auto-active proof of red-black trees in SPARK. In Clark W. Barrett, Misty Davies, and Temesghen Kahsai, editors, *NFM 2017*, volume 10227 of *LNCS*, pages 68–83, Berlin, Heidelberg, 2017. Springer.

[33] Jean-François Dufourd. An Intuitionistic Proof of a Discrete Form of the Jordan Theorem Formalized in Coq with Hypermaps. *Journal of Automated Reasoning*, 43(1):19–51, 2009.

[34] Blair Durkee, Daniel T. Welch, and Murali Sitaraman. Experience report: Rapid reengineering of legacy software using Java reflection. Technical Report RSRG-17-02, Clemson University, 2017.

[35] Alexander Faithfull, Jesper Bengtson, Enrico Tassi, and Carst Tankink. Coqoon - an IDE for interactive proof development in Coq. *STTT*, 20(2):125–137, 2018.

[36] James H. Fetzer. Program verification: The very idea. *Commun. ACM*, 31(9):1048–1063, 1988.

[37] Jean-Christophe Filliâtre. One logic to use them all. In Maria Paola Bonacina, editor, *CADE-24*, volume 7898 of *LNCS*, pages 1–20, Berlin, 2013. Springer.

[38] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *ESOP 2013*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

[39] Carlo A. Furia, Christopher M. Poskitt, and Julian Tschannen. The AutoProof verifier: Usability by non-experts and on standard code. In Catherine Dubois, Paolo Masci, and Dominique Méry, editors, *F-IDE 2015*, 2015.

[40] Lorenzo Gheri and Andrei Popescu. A formalized general theory of syntax with bindings. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *ITP 2017*, volume 10499 of *LNCS*, pages 241–261, Berlin, Heidelberg, 2017. Springer.

[41] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[42] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*, pages 163–179, Cham, 2013. Springer.

[43] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, Quang Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, Thi Hoai An Ta, Nam Trung Tran, Thi Diep Trieu, Josef Urban, Ky Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017.

[44] Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010.

[45] Smith Hampton. *Engineering Specifications and Mathematics for Verified Software*. PhD Dissertation, Clemson University, 2013.

[46] Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Software Eng.*, 17(5):424–435, 1991.

[47] John Harrison. Formalizing an analytic proof of the prime number theorem. *Journal of Automated Reasoning*, 43(3):243–261, 2009.

[48] Heather Harton. *Mechanical and Modular Verification Condition Generation for Object-Based Software*. PhD thesis, Clemson University, 2011.

[49] Martin Hentschel, Stefan Käsdorf, Reiner Hähnle, and Richard Bubel. An interactive verification tool meets an IDE. In Elvira Albert and Emil Sekerinski, editors, *IFM 2014*, volume 8739 of *LNCS*, pages 55–70, Cham, 2014. Springer.

[50] Paolo Herms, Claude Marché, and Benjamin Monate. A certified multi-prover verification condition generator. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *VSTTE 2012*, volume 7152 of *LNCS*, pages 2–17, Berlin, Heidelberg, 2012. Springer.

[51] C. A. R. Hoare. An axiomatic basis for computer programming. *ACM*, 12(10):576–580, 1969.

[52] C. A. R. Hoare. The verifying compiler: A grand challange for computing research. In László Böszörményi and Peter Schojer, editors, *JMLC 2003*, volume 2789 of *LNCS*, pages 25–35, Berlin, Heidelberg, 2003. Springer.

[53] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, first edition, 2003.

[54] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In Kazunori Ueda, editor, *APLAS 2010*, volume 6461 of *LNCS*, pages 304–311, Berlin, Heidelberg, 2010. Springer.

[55] Nabil M. Kabbani, Daniel Welch, Caleb Priester, Stephen Schaub, Blair Durkee, Yu-Shan Sun, and Murali Sitaraman. Formal reasoning using an iterative approach with an integrated web IDE. In *F-IDE 2015*, EPTCS, pages 56–71, 2015.

[56] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006*, volume 4085 of *LNCS*, pages 268–283. Springer, 2006.

[57] James Cornelius King. *A Program Verifier*. PhD thesis, Carnegie Mellon University, 1970.

[58] Jason Kirschenbaum. *Investigations in Automating Software Verification*. PhD thesis, The Ohio State University, 2011.

[59] Jason Kirschenbaum, Bruce M. Adcock, Derek Bronish, Hampton Smith, Heather K. Harton, Murali Sitaraman, and Bruce W. Weide. Verifying component-based software: Deep mathematics or simple bookkeeping? In Stephen H. Edwards and Gregory Kulczycki, editors, *ICSR 2009*, volume 5791 of *LNCS*, pages 31–40, Berlin, Heidelberg, 2009. Springer.

[60] Joachim Klein, Christel Baier, Philipp Chrszon, Marcus Daum, Clemens Dubslaff, Sascha Klüppelholz, Steffen Märcker, and David Müller. Advances in symbolic probabilistic model checking with PRISM. In Marsha Chechik and Jean-François Raskin, editors, *TACAS 2016*, volume 9636 of *LNCS*, pages 349–366. Springer, 2016.

[61] Greg Kulczycki. *Direct Reasoning*. PhD thesis, Clemson University, 2004.

[62] Gregory Kulczycki, Hampton Smith, Heather Harton, Murali Sitaraman, William F. Ogden, and Joseph E. Hollingsworth. The location linking concept: A basis for verification of code using pointers. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *VSTTE 2012*, LNCS, pages 34–49, Berlin, Heidelberg, 2012. Springer.

[63] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV 2011*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[64] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[65] Claire Le Goues, K. Rustan M. Leino, and Michał Moskal. The boogie verification debugger (tool paper). In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *SEFM 2011*, LNCS, pages 407–414, Berlin, Heidelberg, 2011. Springer.

[66] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[67] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. *JML Reference Manual*, May 2013. Draft Revision 2344.

[68] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR 2010*, volume 6355 of *LNCS*, pages 348–370, Berlin, Heidelberg, 2010. Springer.

[69] K. Rustan M. Leino. Developing verified programs with Dafny. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *ICSE 2013*, pages 1488–1490. IEEE, 2013.

[70] K. Rustan M. Leino and Michal Moskał. Usable auto-active verification. In Tom Ball, Lenore Zuck, and Natarajan Shankar, editors, *UV 2010*, 2010.

[71] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *TACAS 2010*, volume 6015 of *LNCS*, pages 312–327, Berlin, Heidelberg, 2010. Springer.

[72] K. Rustan M. Leino and Philipp Rümmer. The Dafny integrated development environment. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *F-IDE 2014*, volume 149 of *EPTCS*, pages 3–15. EPTCS, 2014.

[73] Xavier Leroy. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[74] Azriel. Levy. *Basic Set Theory*. Dover Publications, 2002.

[75] Rupak Majumdar and Viktor Kuncak, editors. *Computer Aided Verification (CAV) 2017 Part I*, volume 10426 of *LNCS*. Springer, 2017.

[76] Rupak Majumdar and Viktor Kuncak, editors. *Computer Aided Verification (CAV) 2017 Part II*, volume 10427 of *LNCS*. Springer, 2017.

[77] Nicodemus Mbwambo. *A Well-Designed, Tree-Based, Generic Map Component to Challenge the Progress Towards Automated Verification*. Masters thesis, Clemson University, 2017.

[78] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[79] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[80] Bertrand Meyer. *Object-oriented Software Construction (2nd Ed.)*. Prentice-Hall, Inc., 1997.

[81] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *IEEE Computer*, 42(9):46–55, 2009.

[82] Kabbani Nabil, Joan Krone, and Murali Sitaraman. Experimentation with proving VCs using no special theory solvers. Technical Report 15-02, Clemson University, 2015.

[83] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[84] Ulf Norell. Dependently typed programming in Agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *AFP 2008*, volume 5832 of *LNCS*, pages 230–266. Springer, 2008.

[85] Nicola Olivetti and Ashish Tiwari, editors. *International Joint Conference on Automated Reasoning (IJCAR) 2016*, volume 9706 of *LNCS*. Springer, 2016.

[86] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *CADE-11*, volume 607 of *LNCS*, pages 748–752, Berlin, Heidelberg, 1992. Springer.

[87] David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976.

[88] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[89] Terrence Parr and Sam Harwell. Another Tool for Language Recognition (ANTLR). Website, available at http://www.antlr.org/.

[90] Amir Pnueli. The temporal logic of programs. In *SFCS 1977*, pages 46–57. IEEE Computer Society, 1977.

[91] Nadia Polikarpova. Eiffelbase2. (repository of verified code) http://toccata.lri.fr/gallery/index.en.html, 2015.

[92] Nadia Polikarpova, Carlo A. Furia, and Bertrand Meyer. Specifying reusable components. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *VSTTE 2010*, volume 6217 of *LNCS*, pages 127–141, Berlin, Heidelberg, 2010. Springer.

[93] Nadia Polikarpova, Carlo A. Furia, Yu Pei, Yi Wei, and Bertrand Meyer. What good are strong specifications? In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *ICSE '13*, pages 262–271. IEEE, 2013.

[94] Nadia Polikarpova, Carlo A. Furia, and Scott West. To run what no one has run before: Executing an intermediate verification language. In Axel Legay and Saddek Bensalem, editors, *RV 2013*, volume 8174 of *LNCS*, pages 251–268, Berlin, Heidelberg, 2013. Springer.

[95] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. A fully verified container library. In Nikolaj Bjørner and Frank S. de Boer, editors, *FM 2015*, volume 9109 of *LNCS*, pages 414–434, Berlin, Heidelberg, 2015. Springer.

[96] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74. IEEE Computer Society, 2002.

[97] Kimberly E. Roche. Mechanical proof checking and its role in establishing software correctness. In Jason O. Hallstrom, editor, *RESOLVE Workshop*, pages 8–13, 2007.

[98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley Professional, 2nd edition, 2010.

[99] Bernd Schoeller. *Making classes provable through contracts, models and frames*. PhD thesis, ETH Zurich, 2007.

[100] Murali Sitaraman, Bruce M. Adcock, Jeremy Avigad, Derek Bronish, Paolo Bucci, David Frazier, Harvey M. Friedman, Heather K. Harton, Wayne D. Heym, Jason Kirschenbaum, Joan Krone, Hampton Smith, and Bruce W. Weide. Building a push-button RESOLVE verifier: Progress and challenges. *Formal Aspects of Computing*, 23(5):607–626, 2011.

[101] Yu-Shan Sun. *Towards Automated Verification of Object-Based Software with Reference Behavior*. PhD thesis, Clemson University, 2018.

[102] Aditi Tagore, Diego Zaccai, and Bruce W. Weide. Automatically proving thousands of verification conditions using an SMT solver: An empirical study. In Alwyn Goodloe and Suzette Person, editors, *NFM 2012*, volume 7226 of *LNCS*, pages 195–209, Berlin, Heidelberg, 2012. Springer.

[103] Toccata. Gallery of verified programs. Website, available at http://toccata.lri.fr/gallery/index.en.html.

[104] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. Autoproof: Auto-active functional verification of object-oriented programs. In Christel Baier and Cesare Tinelli, editors, *TACAS 2015*, volume 9035 of *LNCS*, pages 566–580, Berlin, Heidelberg, 2015. Springer.

[105] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. The AutoProof verifier: Usability by non-experts and on standard code. In *F-IDE 2015*, EPTCS, pages 42–55, 2015.

[106] Alan Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.

[107] Mark Utting, David J. Pearce, and Lindsay Groves. Making Whiley Boogie! In Nadia Polikarpova and Steve Schneider, editors, *IFM 2017*, volume 10510 of *LNCS*, pages 69–84. Springer, 2017.

[108] Willem Visser, Matthew B. Dwyer, and Michael W. Whalen. The hidden models of model checking. *Software and System Modeling*, 11(4):541–555, 2012.

[109] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015.

[110] B. W. Weide, W. F. Ogden, and M. Sitaraman. Recasting algorithms to encourage reuse. *IEEE Software*, 11(5):80–88, Sept 1994.

[111] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In Renate A. Schmidt, editor, *CADE-22*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.

[112] Thomas Wies, Marco Muñiz, and Viktor Kuncak. An efficient decision procedure for imperative tree data structures. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE 2012*, volume 6803 of *LNCS*, pages 476–491. Springer, 2012.

[113] Thomas Wies, Ruzica Piskac, and Viktor Kuncak. Combining theories with shared set operations. In Silvio Ghilardi and Roberto Sebastiani, editors, *FroCoS 2009*, volume 5749 of *LNCS*, pages 366–382. Springer, 2009.

[114] Anna Zaks and Rajeev Joshi. Verifying multi-threaded C programs with SPIN. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *Model Checking Software, SPIN Workshop (2008)*, volume 5156 of *LNCS*, pages 325–342. Springer, 2008.

[115] ETH Zurich/MIT. ComCom/AutoProof - a program verifier for Eiffel. Website, available at http://comcom.csail.mit.edu/comcom/#AutoProof.